# HIGH-PERFORMANCE DOMAIN-SPECIFIC LIBRARY FOR HYDROLOGIC DATA PROCESSING

by

Kalyan Bhetwal


A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2023

BOISE STATE UNIVERSITY GRADUATE COLLEGE

**DEFENSE COMMITTEE AND FINAL READING APPROVALS**

of the thesis submitted by

Kalyan Bhetwal

Thesis Title: High-Performance Domain-Specific Library for Hydrologic Data Processing

Date of Final Oral Examination: 05 May 2023

The following individuals read and discussed the thesis submitted by student Kalyan Bhetwal, and they evaluated the presentation and response to questions during the final oral e xamination. They found that the student passed the final oral examination.

| | |
|---|---|
| Catherine Olschanowsky, Ph.D. | Chair, Supervisory Committee |
| Jim Buffenbarger, Ph.D. | Member, Supervisory Committee |
| Alejandro N. Flores, Ph.D. | Member, Supervisory Committee |

The final reading approval of the thesis was granted by Catherine Olschanowsky, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

Dedicated to my family

# ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Dr. Catherine Olschanowsky, for her guidance, support, and encouragement throughout my research. Her extensive knowledge, experience, and feedback have been invaluable to the successful completion of this thesis.

I would also like to thank the members of my thesis committee, Dr. Jim Buffenbarger and Dr. Alejandro N. Flores, for their insightful comments and suggestions, which have greatly contributed to the improvement of this work.

Finally, I would like to express my deepest gratitude to my family, for their unwavering love, support, and encouragement, without which this thesis would not have been possible.

# ABSTRACT

Hydrologists must process many gigabytes of data for hydrologic simulations, which takes time and resources degrading performance. The performance issues are caused mainly by domain scientists' preference for using Python, which trades performance for productivity. In my thesis, I demonstrate that using the static compilation technique to compile Python to generate C code along with several optimizations reduces time and resources for hydrologic data processing. I developed a Domain Specific Library (DSL) which is a subset of Python and compiles to Sparse Polyhedral Framework - Intermediate Representation (SPF-IR), which allows opportunities for optimizations like read reduction fusion which are not available in Python. We fused the file I/O to perform computation on small chunks of data (stream computation) in order to reduce the memory footprint.

The C code we generated from SPF-IR shows an average speed-up of 2.58x over the existing hand-optimized implementations and can totally eliminate the temporary storage required. DSL users can still enjoy the ease of use of Python but get performance better than the C code.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

**HPC** – High Performance Computing

**DSL** – Domain Specific Language or Library

**SPF** – Sparse Polyhedral Framework

**IR** – Intermediate Representation

**API** – Application Programming Interface

**GB** – Giga Bytes

**I/O** – Input/Output

**HUCs** – Hydrologic Unit Code

**CONUS** – Continent of United States

**PDFG** – Polyhedral Data Flow Graphs

# LIST OF SYMBOLS

$\sqrt{2}$ square root of 2

$\lambda$ lambda symbol, normally used in lambda calculus but it sometimes gets used for wavelength as well

# CHAPTER 1

# INTRODUCTION

## 1.1 Introduction

We use Computational hydrology to understand hydrologic systems better and provide input to various areas including but not limited to agricultural decisions, policy making, and environmental impact studies [1, 2]. Computational hydrology uses large volumes of multidimensional pressure, forcing, and other data collected over a period of time. Pre-processing and post-processing of hydrologic data files using current approaches are too slow. It takes significant time (hours) to complete such processing. The processing is primarily slow due to the software itself. The software for hydrologic data processing is written in Python and developed for convenience and productivity. We propose a solution that minimally changes the development environment while achieving optimized C code performance.

There are several reasons for the considerable time and resources required to process hydrologic data. These programs load large chunks of data into memory for processing, creating a bottleneck in memory utilization and degrading the system's performance. Figure 1.2 shows the existing implementation of hydrologic software. For example, if the file size is 1 GB, we need 1 GB of memory to load that file from the disk for processing. It creates a huge memory footprint. There is a huge gap between memory and CPU speed as shown in figure 1.1. In these computations,

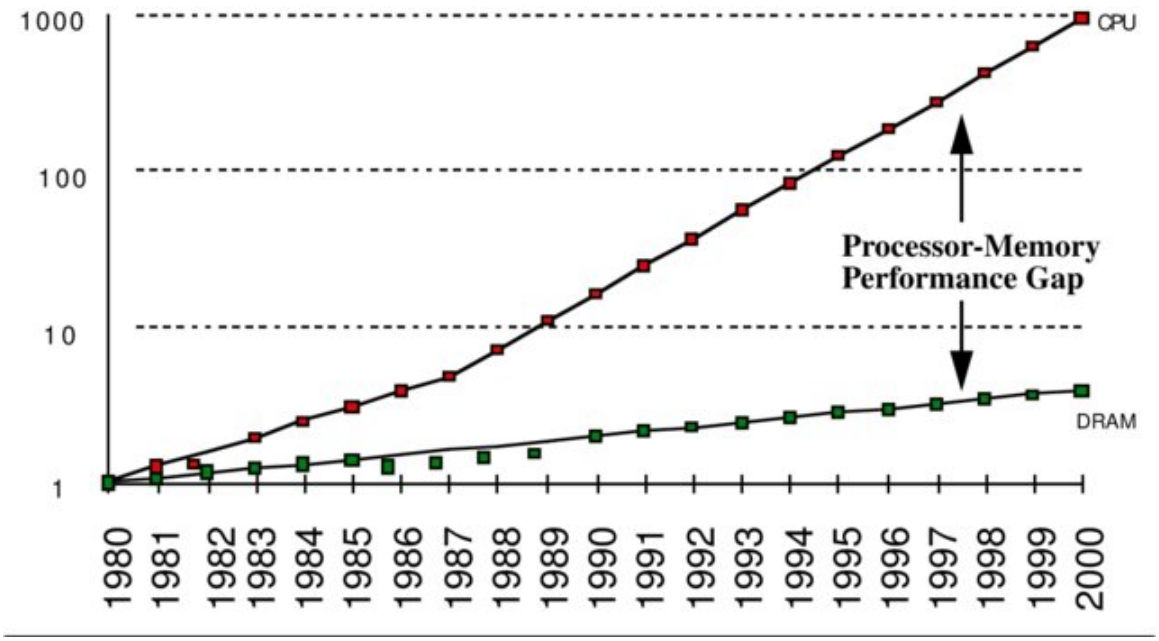


Figure 1.1: CPU-memory performance gap [11]

which process large files, there is a high probability of cache misses, which further degrades the performance.

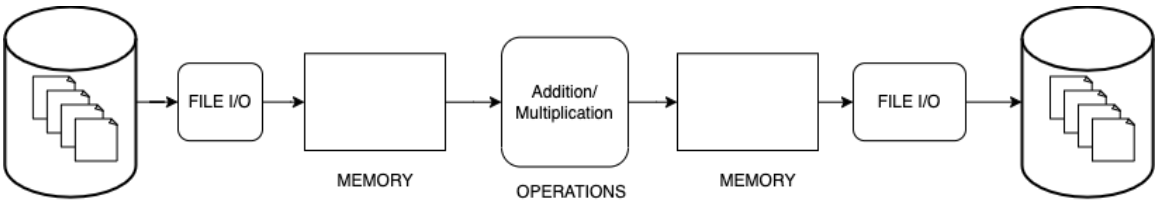


Figure 1.2: Existing Hydrologic system's data flow and memory footprint

Python trades programming performance for productivity. Domain scientists often write data processing software in Python due to its high-level easy-to-use interface. However, this comes at the cost of run-time performance. Although it saves much time for writing codes, it takes significant time for execution. Grosse-Kunstleve, Ralf W., et al. [5] on their work as shown in figure 1.4 demonstrated that the run time performance of Python is worse than compiled languages such as C. The performance

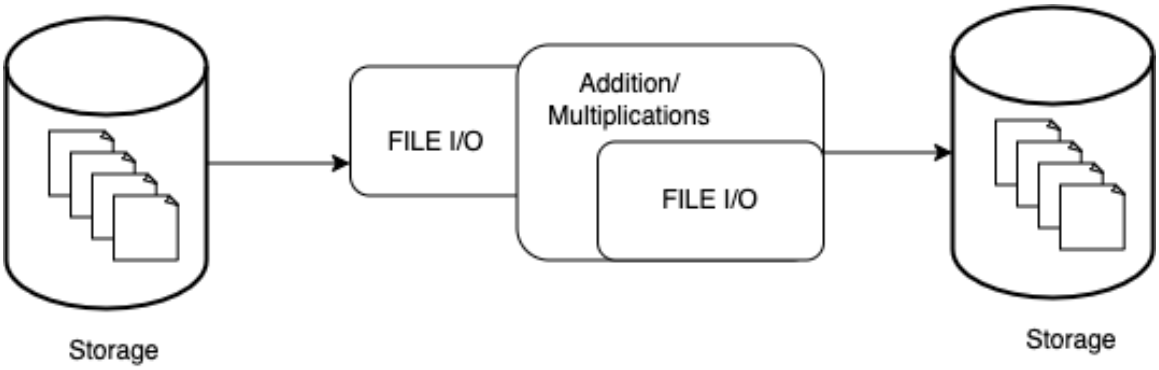


Figure 1.3: Proposed system's data flow and memory footprint

of these programs is crucial because it will save both time and cost in getting those results.

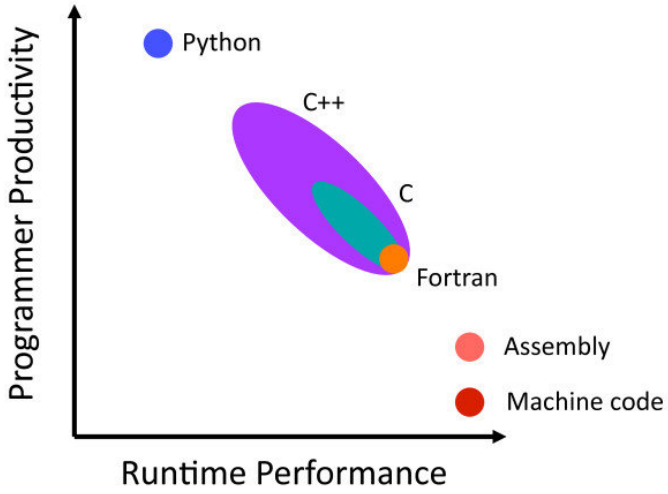


Figure 1.4: Qualitative graph comparing runtime performance and programmers' productivity for different programming languages

Compilation techniques that would improve the performance of these programs exist but can not be used by Python directly. A deeper code analysis shows that we could do optimization, including read reduction fusion and more. These optimizations aren't directly possible in Python. Computer scientists use polyhedral

compilation for performance optimization in high-performance computing [6]. The sparse polyhedral framework (SPF) extends the polyhedral framework to support non-affine loop bounds. IEGenLib is an open-source C++ library to implement the algorithm that manipulates sets and relations with uninterpreted function symbols to enable the Sparse Polyhedral Framework [16]. The Computation API provides an object-oriented interface that wraps SPF Intermediate Representation (IR) [12].

In this work, we present a Domain-Specific Library (DSL) designed to optimize the performance of hydrologic data processing. The DSL is a subset of Python that compiles to the SPF-IR and provides all the necessary functions commonly used in hydrologic data processing. Figure 1.2 shows the data flow for the proposed system. We fuse file I/O with the calculations, which reduces the memory footprint. This allows us to load data from disk to memory optimally, immediately perform computations on them and store the data back to the disk. The Python interface provided to users maintains the same productivity while enjoying the performance of the C programming language.

## 1.2 Research Overview

This thesis explores creating a library for improving the performance of hydrologic data processing while minimizing the impact on the programmers' productivity. Figure 1.3 shows the high-level research overview of the project. The library builds an SPF-IR representation of the desired computation rather than computing it right away. When a command indicates that the calculation must be complete, such as data being output to a file, we apply different optimization in the SPF-IR and generate the C code.

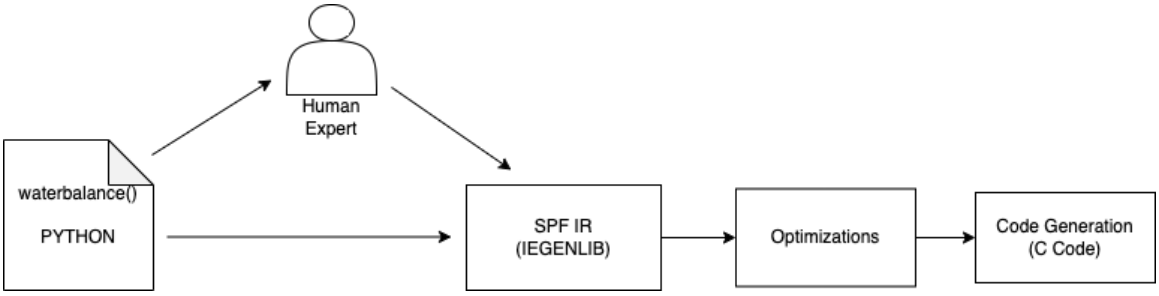


Figure 1.5: Flow chart of the proposed system

Our research contributions can be summarized into three points below:

***Contribution 1:*** *Designed a pipeline in Python that reflects the current PFtools API. This pipeline included I/O.*

***Contribution 2:*** *Implemented a collection of common hydrological data processing procedures in the pipeline.*

***Contribution 3:*** *Transformed and evaluated the performance of the optimized pipelines. This included an evaluation of the overhead of delayed execution.*

## 1.3 Contribution 1: Designed a pipeline in Python that reflects the current PFtools API. This pipeline included I/O.

A pipeline in our library is a collection of computations that an end user will perform for a particular task. The pipeline starts with file input and exits when we write the file back to the disk.

## 1.4 Contribution 2: Implemented a collection of common hydrological data processing procedures in the pipeline.

We implemented common data processing procedures like average and sum. The procedures are exposed through the Python library.

## 1.5 Contribution 3: Transformed and evaluated the performance of the optimized pipelines. This included an evaluation of the overhead of delayed execution.

We generated the C code from the Python library. The C code has file input fused with the computations. The generated C code is compiled and executed.

# CHAPTER 2

# BACKGROUND

In this section, we discuss various topics needed to understand this work. We are taking PFtools as the running example throughout this thesis to test our hypothesis. For optimizations, we represent the PFtools codebase in the sparse polyhedral framework using the Computation API.

## 2.1 PFtools

PFtools are a collection of tools for pre-processing and post-processing of data for ParFlow [9] simulations. Domain scientists use Python in PFtools for convenience, which comes at a performance cost. PFTools read parflow binary (PFB) files as input and write back the PFB file after the simulations are completed. Figure 4 shows a hydrologic workflow showing different steps. PFB files store pressure, temperature, humidity, and more information for the whole continent of the United States (CONUS). We have an area of interest called hydrologic Units (HUCs) for running the simulation. We do subsetting to extract data for a particular area of interest. The Parflow Simulation Software uses the result of subsetting. The Parflow is massively parallel and highly efficient which is typically run in supercomputers. But, the PFtools applications are written in Python and are not optimized. This thesis

revolves around improving the performance of the software for pre and post-processing data surrounding the Parflow simulation.

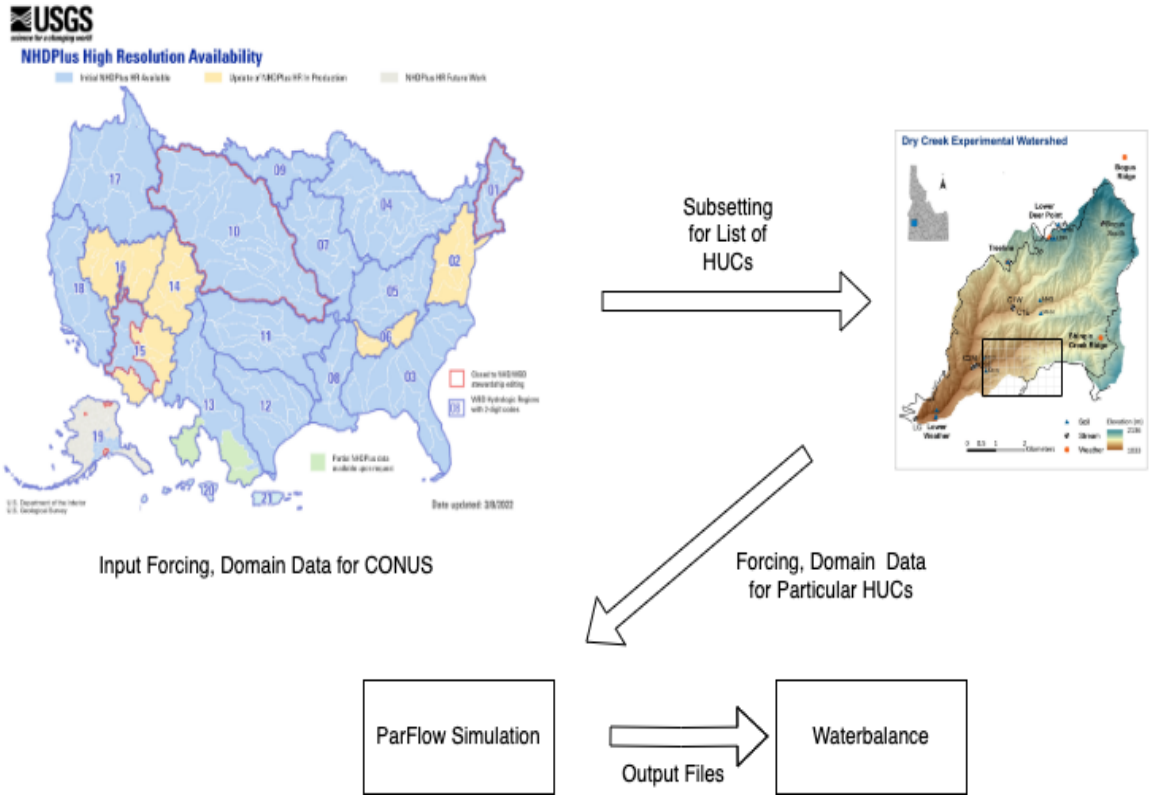


Figure 2.1: A workflow for Water Balance Calculation

## 2.2 Polyhedral Model

Scientific computations involve iterating over large data spaces using nested loops for computations. These simulations can largely benefit from parallelism and data locality. A polyhedral framework is a mathematical representation of loops for various automatic loop transformations and optimizations.

Consider an example below, which has affine loop bounds.

```
for(int i=0;i<M ;i++){
    for(int j=0; j<N;j++){
       S0: s(i,j) = x(i)+j;
    }
}
```

The Polyhedral representation of the computation is given as:

$$S0 : s(i,j) = x(i) + j;$$

$$D0 : I = \{[i,j] : 0 \leq i < M \ \wedge \ 0 \leq j < N\} \quad E0 : \{[i,j] \rightarrow [0,i,0,j,0]\}$$

$$R0 : \{[i,j] \rightarrow [i]\}$$

$$W0 : \{[i,j] \rightarrow [i,j]\}$$

The execution of loops of the above program can be represented in the Polyhedral model as a set of all the valid combinations of tuple $[i,j]$.

$$I = \{[i,j] : 0 \leq i < M \ \wedge \ 0 \leq j < N\}$$

The iteration space combined with data dependence provides partial order of the execution of the given computations. To change the execution order, we apply relations to the iteration space. We can express a relation for loop interchange for the above example as:

$$R = \{[i,j] \rightarrow [j,i]\}$$

$$I' = R(I)$$

The code synthesis after the above transformation yields the following program.

```
for(int j=0;i<M ;i++){
    for(int i=0; j<N;j++){
       S0: s(i,j) = x(i) +j;
    }
}
```

The polyhedral framework makes it simpler to go from the code to the mathematical representation and back to the code again after applying transformations. The polyhedral framework is limited to affine data access. However, sparse data accesses are non-affine in nature. The sparse polyhedral framework overcomes the limitations of the polyhedral framework by supporting non-affine constraints in iteration space as *uninterpreted functions*.

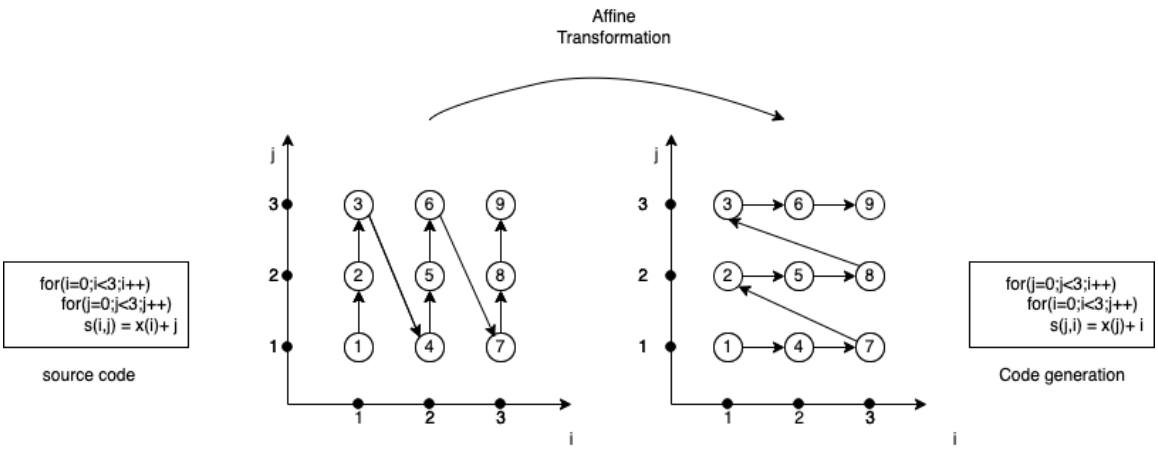


Figure 2.2: Polyhedral Framework

### 2.2.1 Iteration Domain

The polyhedral framework represents each point in these nested loops as a lattice point in polyhedra. Figure 2.2 shows the polyhedral representation of a code. Each

point in the lattice is represented as a set. A relation is applied to the set for the transformations.

### 2.2.2 Scheduling

A schedule provides lexicographical ordering of a statement. It is represented as a relation. The dimension of the schedule gives the number of loop nests. We can do affine transformation without violating the data dependence relationship after the polyhedral representation. The transformations can help in parallelism and data locality by finding the best schedules for those loops.

## 2.3 Computation API

The Computation API is an object-oriented interface to SPF IR [12]. The API provides functionality to interact with the SPF and generates the polyhedral data flow graphs (PDFGs). The computation API integrates CHiLL [4], CodeGen+ [3], Omega [8], and IEGenLib [16]. The API provides a computation class to express a computation or series of computations. This work creates a computation object containing data spaces, statements, data dependencies, and execution schedules for one specific computation. Figure 5 shows the implementation of Sparse vector multiplication in the Computation API.

### 2.3.1 Code Generation

Code generation is another vital functionality available in the computation API. It is the final and most crucial stage where actual code is generated for the given computation specification using CodeGen+. The Codegen+ uses omega for polyhe-

Sparse matrix-vector multiplication

```
/*Sparse vector multiply
for (i = 0; i < N; i++) {
    for (j=rowptr[i]; j<rowptr[i+1];j++) {
        k = col[j];
        y[i] += A[j] * x[k];
}}*/
Computation* sparseComp = new Computation();
sparseComp->addDataSpace("y", "int*");
sparseComp->addDataSpace("A", "int*");
sparseComp->addDataSpace("x", "int*");
Stmt* sparseS0 = new Stmt(
  "y(i) += A(k) * x(k)", // Source code
  // iteration domain
  "{[i,j,k]: 0<=i<N && rowptr(i)<=j<rowptr(i+1) && k=col(j)}",
  "{[i,j,k]->[0,i,0,j,0,k,0]}", // Scheduling Function
  { {"y", "{[i,j,k]->[i]}"},
    {"A", "{[i,j,k]->[j]}"},
    {"x", "{[i,j,k]->[k]}"}}, // Data reads
  { {"y", "{[i,j,k]->[i]}"} } // Data writes
);
sparseComp->addStmt(sparseS0);
```

Figure 2.3: Sparse matrix-vector multiplication representation in the Computation API.

dra scanning. Omega has certain limitations where it cannot handle uninterpreted functions. Codegen+ overcomes this limitation by altering the uninterpreted function in IEGenLib to be Omega Compliant.

## 2.4 Polyhedral Dataflow Graphs

A Polyhedral Dataflow Graph (PDFG) is a visual representation in the form of a graph for data dependencies between a series of computations [15]. PDFG helps performance engineers to make various optimization decisions like dead code elimination as it provides a comprehensive view of data flow.

In this thesis, we use PDFG to analyze the dataflow patterns in the PFTools API to make optimization decisions. Figure 2.4 shows the Matricized Tensor Times Khatri-Rao Product (MTTKRP). We can generate the PDFG for the computation (shown in Figure 2.5) which enables us the view the dataflow and make optimization decisions on them.

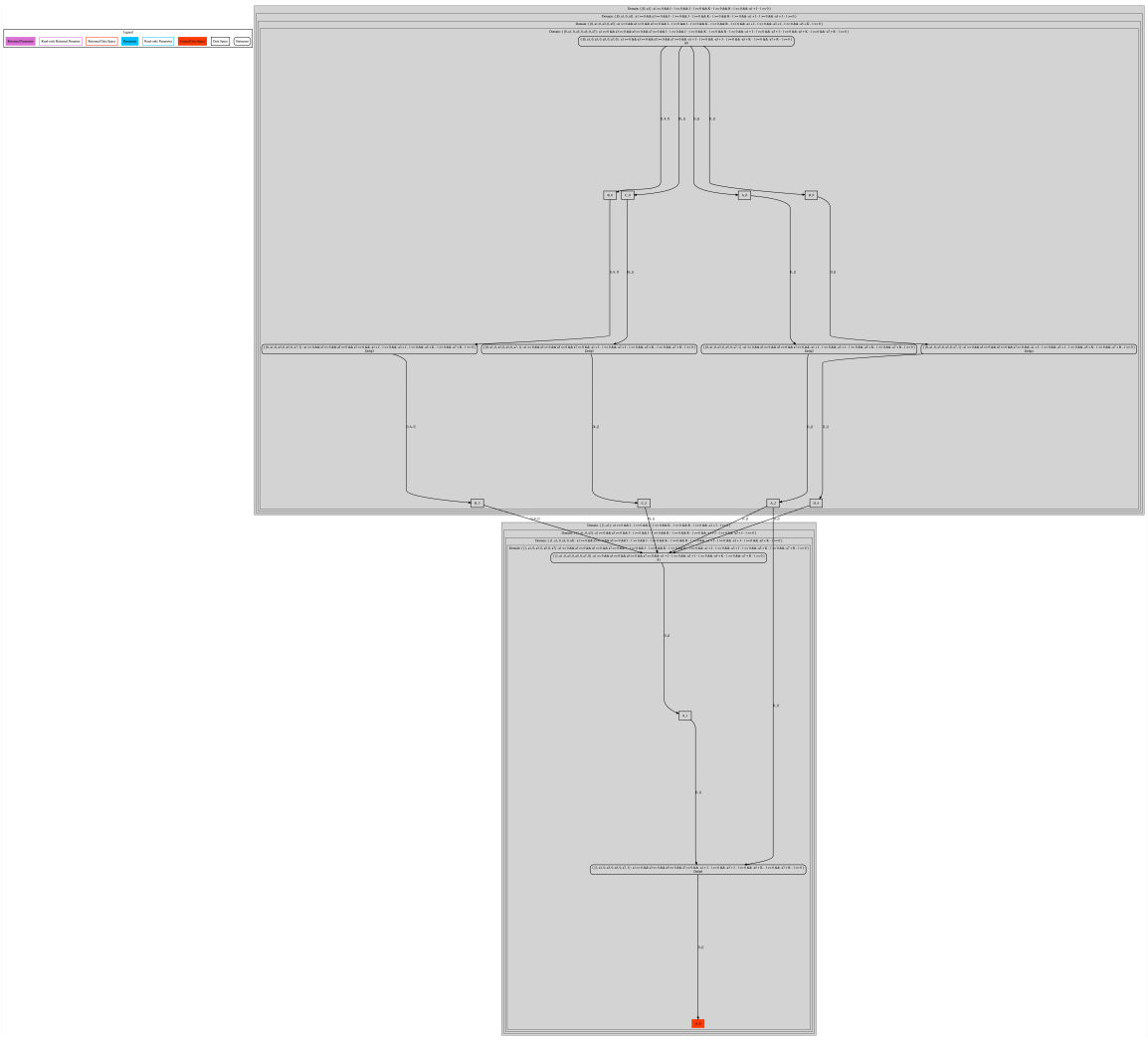

Figure 2.4: PDFG for MTTKRP

Sparse matrix-vector multiplication

```
/*MTTKRP
    for (i = 0; i < I; i++)
        for (j = 0; j < J; j++)
          for (k = 0; k < K; k++)
            for (r = 0; r < R; r++)
              A[i,r] += X[i,j,k]*B[j,r]*C[k,r];
*/
Computation* sparseComp = new Computation();
sparseComp->addDataSpace("y", "int*");
sparseComp->addDataSpace("A", "int*");
sparseComp->addDataSpace("x", "int*");
Stmt* sparseS0 = new Stmt(
  "y(i) += A(k) * x(k)", // Source code
  // iteration domain
  "{[i,j,k]: 0<=i<N && rowptr(i)<=j<rowptr(i+1) && k=col(j)}",
  "{[i,j,k]->[0,i,0,j,0,k,0]}", // Scheduling Function
  { {"y", "{[i,j,k]->[i]}"},
    {"A", "{[i,j,k]->[j]}"},
    {"x", "{[i,j,k]->[k]}"}}, // Data reads
  { {"y", "{[i,j,k]->[i]}"} } // Data writes
);
sparseComp->addStmt(sparseS0);
```

Figure 2.5: MTTKRP multiplication representation in the Computation API.

# CHAPTER 3

# PFTOOLS API PIPELINE

PFtools follow common patterns for processing data files. They take the PFB file of large sizes (> 2.6 GB) as input. Parflow binary files store crucial pieces of information about pressure, temperature, and humidity. The file is in binary format. It contains 3-dimensional data organized in a rectangular block-like structure. The start of each block in the pfb file contains information about the starting x,y, and z coordinates and dimensions of the block. Figure 3.2 shows the data access pattern. The data is read one block at a time. Each block dimension is nx * ny * nz given by the block header information available in the file. The data is continuous along the x-axis in memory so any computations that use this data would benefit from cache locality if the computation reads continuously along the x-axis.

The input file is fully read and loaded into the memory. The computations like sum and average calculations are performed. We store the results back to disk to be further used in other calculations. Fig 3.1 illustrates the complete structure of a pipeline for average calculation. We can break down a pipeline into three specific parts: read, compute, and store.

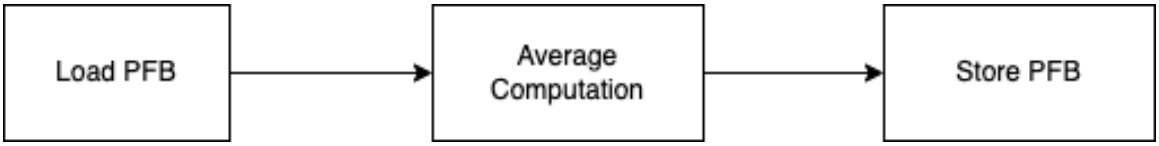


Figure 3.1: A PFTools API Pipeline for Computing Mean

## 3.1 Reading PFB from Disk

The first computation in the pipeline is reading data from the disk. Listing 3.1 shows the snippet of the code for loading data into memory. The outer loop goes through each block of the data. The inner 3 loops read data from those grids and store the data in the "m_data" array.

```
for (nsg = 0;nsg<m_numSubgrids; nsg++){
    for (k=0; k<nz; k++){
          for(i=0;i<ny;i++){
               for(j=0;j<nx;j++){
                     m_data[index+ j] = tmp;
               }
          }
      }
}
```

Listing 3.1: Loops to read data from the file

## 3.2 Computations

We perform several computations of the data obtained from the parflow binary files. Some of the computations are average, sum, max, and min.

In this thesis, we discuss average computation as an example.

### 3.2.1 Average

Consider listing 3.2 which is used to compute the mean of the data along the z-axis. As we discussed in an earlier section 3.1, the data is continuous along the x-axis. But

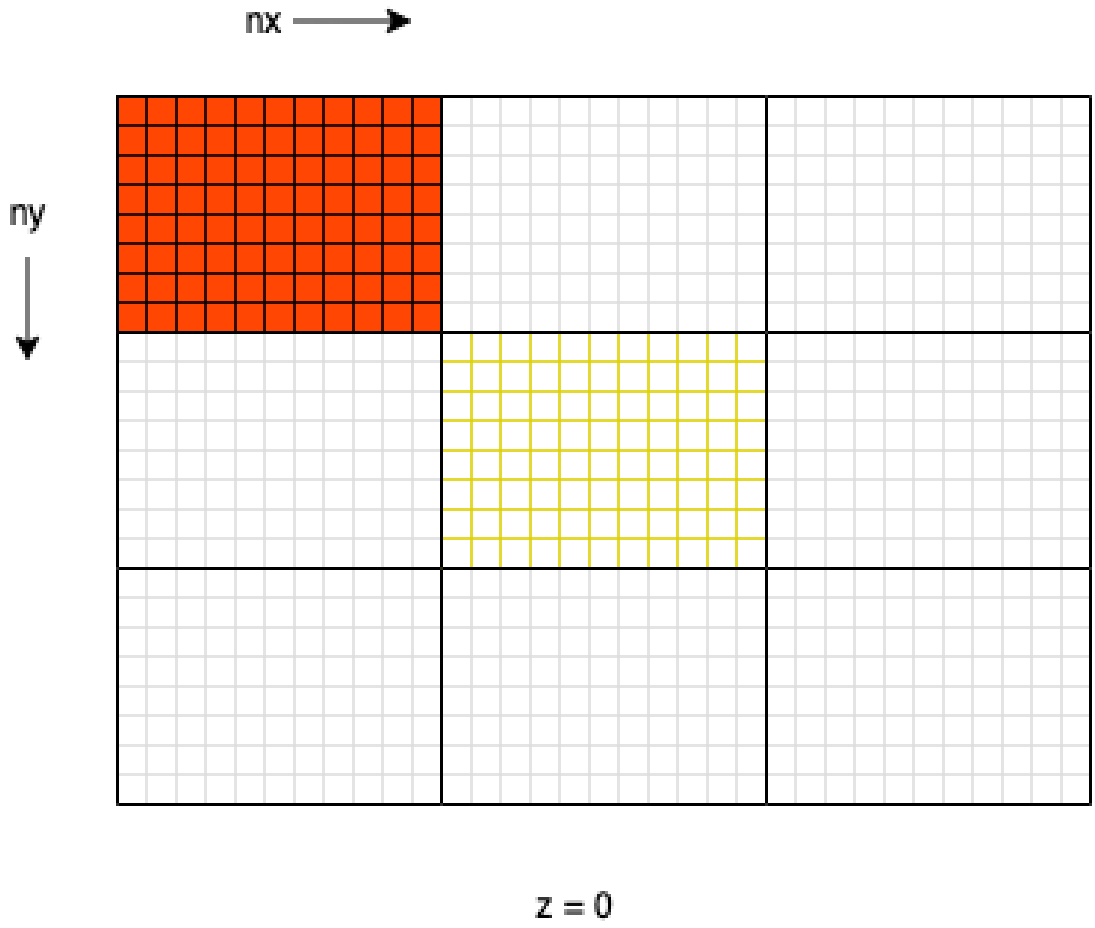


Figure 3.2: Organization of data in the grid pattern

the innermost loop is the mean along the z-axis and runs continuously on the z-axis. This creates lots of jumps in memory access patterns. While reading data from the disk, we read the data in smaller block sizes but while computing the mean we are using the larger chunk of data.

```
1 for(int x = 0; x < m_nx; x++){
2    for(int y = 0; y < m_ny; y++){
3        sum = 0;
4         for(int z = 0; z < m_nz; ++z){
5               sum +=m_data[static_cast<long long>(z)*m_ny*m_nx+y*m_nx+x];
6        }
7        mean[(x+y*m_nx)] = sum/m_nz
```

```
8     }
9 }
```

Listing 3.2: Section of code to compute mean

## 3.3 Writing PFB to Disk

Once the computations are performed the data is written back to the disk in the parflow binary formats for further use in the simulation. The read-and-write process is similar and consistent. Section 3.3 shows the section of code to write data to disk. We iterate over the z,y, and x dimensions to write data back.

```
1 for(iz=calcOffset(m_nz,m_r,nsg_z); iz < calcOffset(m_nz,m_r,nsg_z+1);iz++){
2   for(iy=calcOffset(m_ny,m_q,nsg_y); iy < calcOffset(m_ny,m_q,nsg_y+1);iy++){
3         uint64_t* buf = (uint64_t*)&(m_data[iz*m_nx*m_ny+iy*m_nx+calcOffset(
              m_nx,m_p,nsg_x)]);
4         long long j;
5         for(j=0;j<x_extent;j++){
6             uint64_t tmp = buf[j];
7             tmp = bswap64(tmp);
8             writeBuf[j] = *(double*)(&tmp);
9         }
10        fwrite(writeBuf.data(),sizeof(double),x_extent,fp);
11    }
12 }
```

Listing 3.3: Section of code to write data back to the disk

## 3.4 Chapter Summary

From the codes in listing 3.1, 3.2, and 3.3 we can see that we are iterating over x, y, and z dimensions time and again on the same data for our computations. It makes sense to represent these programs in a Polyhedral framework to reason about them and apply different transformations for the optimized program. In the next chapter, we discuss representing these computations in SPF and transforming them.

# CHAPTER 4

# THE LIBRARY/LANGUAGE

Users are provided with a Python library that contains functions commonly present in PFtools API. The library consists of a collection of functions needed for a hydrologic data processing tool. Each function consists of a series of computations. These functions are exposed through a library.

The library is a subset of Python that can be used in any Python interpreter. When an end user writes a program for e.g. to compute a mean and runs it in the Python interpreter, no operations happen till the point where the user requests a file output. During this time we just record the operation that the user is trying to perform. As soon as we reach the point where the actual I/O is requested, a series of optimization recipes are applied. The main optimization recipe is file I/O fusion. We use a heuristic to determine the best block size of data that is loaded into memory from the disk which is then used in the computation. As soon as we compute the results for the loaded block we store it back on the disk. After we are done with the optimizations we spit out the C++ code. This code is then compiled with any C++ compiler and is used for executing the user program.

## 4.1 The Library from End User Perspective

End users are exposed to a series of functions through a library written in Python. This library wraps the Computation API which allows all the functionality to represent the necessary computations in the Sparse Polyhedral framework along with the functionality for code generation. Users can import the library and call functions as per their needs. Each time a function is called the library just keeps the track of functions and at the end when the output is requested, the library combines all the functions and generates the C code after applying a series of optimizations.

```
import iegenlib
iegenlib.readFile("snakeriver.pfb")
iegenlib.compute_mean(axis="z")
iegenlib.storeFile("snakeriver.pfb")
```

Listing 4.1: Program to Compute mean

## 4.2 Computation API Representation

In the previous chapter, we identified all the potential functions that need to be implemented for hydrologic data processing tools. The first stage for our library is to represent these functionalities in the Computation API. Consider a small section of code that is currently used for the mean computation of a file along the z-axis.

```
1 for(int x = 0; x < m_nx; x++){
2     for(int y = 0; y < m_ny; y++){
3         mean[x+m_nx*y] = 0;
4         for(int z = 0; z < m_nz; z++){
5                 mean[x+m_nx*y] +=m_data[static_cast<long long>(z)*m_ny*m_nx+y*
                      m_nx+x];
6         }
7         mean[(x+y*m_nx)] = mean[x+m_nx*y] /m_nz
8     }
9 }
```

The corresponding Computation API representation is:

```
dataReads1 = iegenlib.PairVector([])
dataWrites1 = iegenlib.PairVector([("mean","{[x,y]->[x,y]}")])
s1 = iegenlib.Stmt("mean[x+m_nx*y] = 0;",
                   "{[x,y]:0<=y<m_ny && 0<=x<m_nx}",
                   "{[x,y]->[0,x,0,y,1]}",
                   dataReads1,
                   dataWrites1)
  parflowio_mean.addStmt(s1)


  dataReads2 = iegenlib.PairVector([("m_data","{[x,y,z]->[z,y,x]}")])
  dataWrites2 = iegenlib.PairVector([("mean","{[x,y,z]->[x,y]}")])

  s2 = iegenlib.Stmt("mean[x+m_nx*(y)]+=m_data[(long long)(z)*m_ny*m_nx+(y)*
      m_nx+x];",
                   "{[x,y,z]:0<=y<m_ny && 0<=x<m_nx && 0<=z<m_nz}",
                   "{[x,y,z]->[0,x,0,y,1,z,0]}",
                   dataReads2,
                   dataWrites2)
  parflowio_mean.addStmt(s2)

  dataReads3 = iegenlib.PairVector([("mean","{[y,x]->[y,x]}")])
  dataWrites3 = iegenlib.PairVector([("mean","{[y,x]->[y,x]}")])

  s3 = iegenlib.Stmt("mean[x+m_nx*y] = mean[x+m_nx*y]/m_nz;",
                   "{[x,y]:0<=y<m_ny && 0<=x<m_nx}",
                   "{[x,y]->[0,x,0,y,2]}",
                   dataReads3,
                   dataWrites3)
  parflowio_mean.addStmt(s3)
```

In order to represent each statement in the original program in Computation API, we need four specific components.

- Statement: *The statement portion contains the actual statement string in the source code*
- Iteration Space: *The iteration space provides iteration domain of the loop nests*
- Execution Schedule: *The execution schedule gives us the order of the execution of statements*

- Data Reads: *Data reads writes provide the data read relations*
- Data Writes: *Data writes provide the data read relations*

## 4.3 Functions

The functions provided in our library are the Python function that performs one particular task in the PFtools API. Each function contains a series of statements that altogether form a computation. The function starts with creating a computation object and returns the computation object.

```
def readFile(filename):
    comp = iegenlib.Computation()
    comp.addStmt(s0)
    comp.addStmt(s1)
    return comp
```

## 4.4 Append Computations

When we combine two or more functions for a complete program, we need to append all the computations to form a single computation. The execution schedule needs to adjust to form a new single computation with increasing order of execution schedule.

```
comp1 = readFile("snakeriver.pfb")
comp2 = compute_mean(axis='z')
comp3 = storeFile("snakeriver.pfb")
```

The Computation API provides the function to append computation to another. Consider if the execution schedule of "comp1" ends with a relation "$[0] \rightarrow [3]$". After we append the comp2 to comp1 the execution schedule of the first statement should start with "$\{[0] \rightarrow [4]\}$". We can achieve this by using the command below:

```
comp1.appendComputation(comp2, "{[0]}","{[0]->[4]}")
```

In the appendComputation function, the first argument is the computation that needs to be appended, the third argument takes the new execution schedule from which the new execution schedule should start for comp2 in combined computation.

## 4.5 Optimization Recipes

We have the opportunity to optimize the program in order to reduce memory use and run time. The library also provides a function to apply a series of optimizations. We explored the code at the first and found that we can re-order the loops for better data access patterns. We also can fuse multiple loops to combine loops with producer-consumer patterns.

### 4.5.1 Loop Transformation

In loop transformation, we change the order of the loop nesting. The outer loop can be moved to the inner part. The main objective of the loop transformation is to improve the data access pattern such that maximum data can be reused from the cache.

```
1 import iegenlib
2
3 parflowio = iegenlib.Computation()
4 dataReads4 = iegenlib.PairVector([("m_data","{[x,y,z]->[z,y,x]}")])
5 dataWrites4 = iegenlib.PairVector([("mean","{[x,y,z]->[x,y]}")])
6
7 s4 = iegenlib.Stmt("mean[x+y*m_nx]+=m_data[(long long)(z)*m_ny*m_nx+y*m_nx+x];
      ",
8                   "{[x,y,z]:0<=y<m_ny && 0<=x<m_nx && 0<=z<m_nz}",
9                   "{[x,y,z]->[0,x,0,y,1,z,0]}",
10                  dataReads4,
11                  dataWrites4)
12 parflowio.addStmt(s4)
13
14 print(parflowio.codeGen())
```

Listing 4.2: Section of Computation specification to calculate mean

The generated code from the above computation specification is:

```
#undef s0
#undef s_0
#define s_0(x, y, z) mean[x+y*m_nx]+=m_data[(long long)(z)*m_ny*m_nx+y*m_nx+x
    ];
#define s0(__x0, x, __x2, y, __x4, z, __x6) s_0(x, y, z);


int t1 = 0;
int t2 = 0;
int t3 = 0;
int t4 = 0;
int t5 = 1;
int t6 = 0;
int t7 = 0;

if (m_nz >= 1 && m_ny >= 1) {
  for(t2 = 0; t2 <= m_nx-1; t2++) {
    for(t4 = 0; t4 <= m_ny-1; t4++) {
        for(t6 = 0; t6 <= m_nz-1; t6++) {
        s0(0,t2,0,t4,1,t6,0);
      }
    }
  }
}

#undef s0
#undef s_0
```

Listing 4.3: The generated code for computing the mean

We have already established in chapter 3 that the data is continuous in the x-direction. The generated code in accessing the data in the z-direction first. So, there will be many jumps in the memory to access the data. We could benefit from caching if we can read in the x-direction. We use loop transformation for this. The Computation API provides an easy interface for performing loop transformation. We need to define a relation that can map our initial execution schedule to a target execution schedule. For our example, to change the loop order of loops from [x,y,z] to [z,y,x] we need to apply the below relation.

```
rel = iegenlib.Relation("{[0,x,0,y,0,z,0]-> [0,z,0,y,0,x,0]}")
parflowio.addTransformation(stmtIndex=0,rel=rel)
```

Listing 4.4: Relation and function for the loop transformation

In listing 4.4 we can see the transformed loops where the x-direction is read first which is the desired outcome.

```
15 #undef s0
16 #undef s_0
17 #define s_0(x, y, z) mean[x+y*m_nx]+=m_data[(long long)(z)*m_ny*m_nx+y*m_nx+x
       ];
18 #define s0(__x0, z1, __x2, y1, __x4, x1, __x6) s_0(x1, y1, z1);
19
20
21 int t1 = 0;
22 int t2 = 0;
23 int t3 = 0;
24 int t4 = 0;
25 int t5 = 0;
26 int t6 = 0;
27 int t7 = 0;
28
29 if (m_nx >= 1 && m_ny >= 1) {
30   for(t2 = 0; t2 <= m_nz-1; t2++) {
31     for(t4 = 0; t4 <= m_ny-1; t4++) {
32         for(t6 = 0; t6 <= m_nx-1; t6++) {
33           s0(0,t2,0,t4,0,t6,0);
34       }
35     }
36   }
37 }
38
39 #undef s0
40 #undef s_0
```

Listing 4.5: The generated code after loop transformation

### 4.5.2 Tiling

To perform the fusion between the file reading portion and mean computation we need to tile at first. The upper bounds of the loops for the reading file and mean computation are different. We break loops for mean computation in tiles of the size identical to loops of the file reading portion of the code. Consider listing 4.5 and 4.6.

Listing 4.5 is used for reading data from the file while listing 4.6 is used for computing the mean. If we re-order the loops in listing 4.6 to match the looping in listing 4.5 i.e if both listings have the same loop nest "[z,y,x]", then we can tile and fuse them.

Listing 4.5 and 4.6 are iterating over the same domain of data. In listing 4.5 only a block of data is being considered at a time whereas listing 4.6 considers the whole data. Establishing the domain of data for listing 4.5 and 4.6 is a separate work in itself. For our work, we consider they are equivalent.

```
42 for(t2 = 0; t2 <= m_numSubgrids-1; t2++) {
43   s3(3,t2,0);
44   s4(3,t2,1);
45   if (ny >= 1) {
46   for(t4 = 0; t4 <= nz-1; t4++) {
47   for(t6 = 0; t6 <= ny-1; t6++) {
48         s5(3,t2,2,t4,0,t6,0);
49   for(t8 = 0; t8 <= nx-1; t8++) {
50           s6(3,t2,2,t4,0,t6,1,t8,0);
51         }
52       }
53     }
54   }
55 }
```

Listing 4.6: Read Data

```
1  for(t2 = 0; t2 <= m_ny-1; t2++) {
2  for(t4 = 0; t4 <= m_nx-1; t4++) {
3      s8(4,t2,0,t4,1);
4  for(t6 = 0; t6 <= m_nz-1; t6++) {
5        s9(4,t2,0,t4,1,t6,0);
6      }
7      s10(4,t2,0,t4,2);
8    }
9  }
```

Listing 4.7: Compute Mean

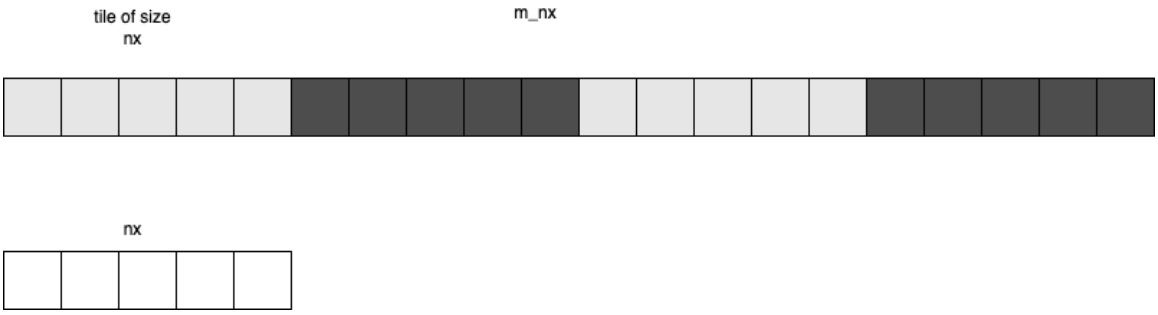


Figure 4.1: Block of size m_nx tiled into size of nx

### 4.5.3 Producer-Consumer Loop fusion

In Loop fusion, we combine two or more statements in different loop nests into single loop nests. The data produced in the Listing 4.5 can be consumed directly by Listing 4.6. The outcome of the producer-consumer loop fusion is:

```
1  for(t2 = 0; t2 <= m_numSubgrids-1; t2++) {
2    s3(3,t2,0);
3    s4(3,t2,1);
4    if (ny >= 1) {
5      for(t4 = 0; t4 <= nz-1; t4++) {
6        for(t6 = 0; t6 <= ny-1; t6++) {
7          s5(3,t2,2,t4,0,t6,0);
8          for(t8 = 0; t8 <= nx-1; t8++) {
9  s6(3,t2,2,t4,0,t6,1,t8,0);
10 s9(3,t2,2,t4,0,t6,1,t8,1);
11         }
12       }
13     }
14   }
15 }
```

Listing 4.8: Fusing input and mean computation

## 4.6 Code Generation

The Computation API also provides the functionality for code generation from the computation object. We generate the C code as soon as we have a complete pipeline

i.e. a user requests output. The code is then compiled with any standard C++ standard compiler.

# CHAPTER 5

# RESULTS

## 5.1 Experimental Setup

All the experiments were run on the verde server hosted at Princeton University. The verde server is an HPC cluster with 96 CPUs of model AMD EPYC 7402 24-Core Processor. The 96 CPUs are distributed in 16 numa nodes with each node consisting of 6 CPUs.

All the generated codes are compiled with GCC compiler version 8.5.0 20210514.

## 5.2 Performance Evaluation

We evaluate the performance of our approach with existing Python-based implementation. We selected five PFB files of size 2.6 GB for benchmarks. We recorded the time to compute the mean for the existing approach and our implementation.

### 5.2.1 Execution Time

Table 5.1 shows the execution time for both of the implementations. We can clearly see that our approach is much faster than the existing implementation with an average speedup of 2.58x. Figure 5.1 shows the average execution time for both of the implementations.

There is overhead in generating the C-code, compiling, and running them. Since these programs are run time and again the code of delayed execution (codegen and execution) is amortized.

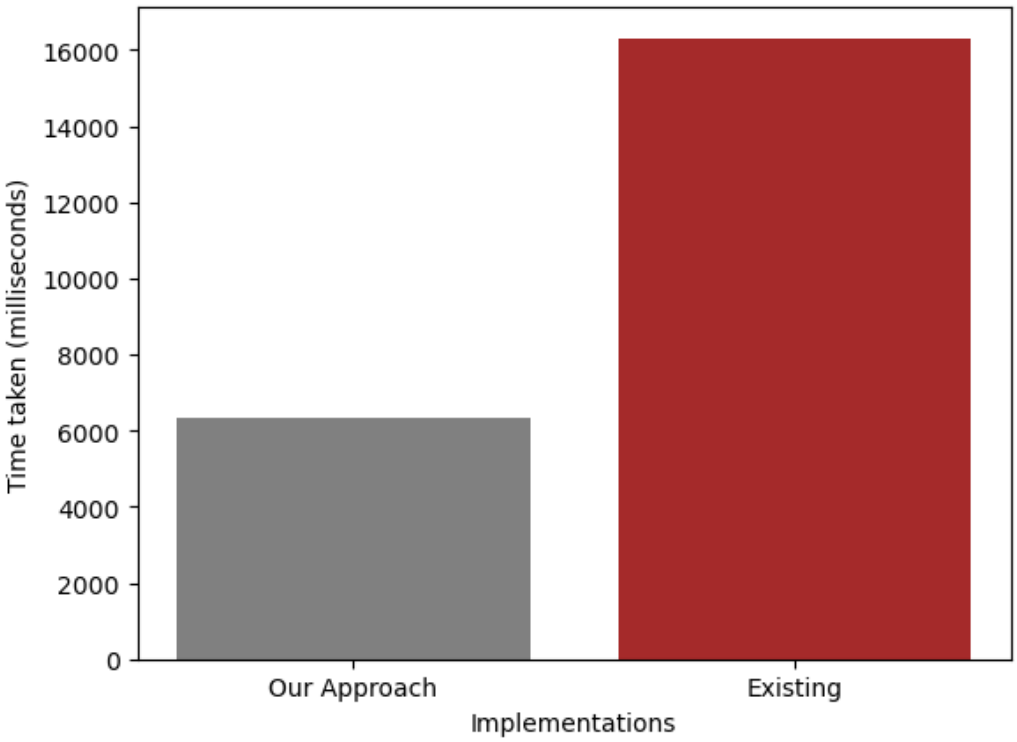


Figure 5.1: Comparions of execution times of our implementation vs existing implementation

### 5.2.2 Memory Utilization

With our approach, we could totally eliminate the temporary storage required for the memory footprint as we are immediately performing computations of the block of data that is being read from the file.

| File | Our Approach(ms) | Existing(ms) | Speedup |
|---|---|---|---|
| NLDAS.Press.001777_to_001800.pfb | 6254 | 11206 | 1.79 |
| NLDAS.Press.000433_to_000456.pfb | 6427 | 16401 | 2.55 |
| NLDAS.Press.000697_to_000720.pfb | 6543 | 17011 | 2.60 |
| NLDAS.Press.000841_to_000864.pfb | 6459 | 19354 | 3.00 |
| NLDAS.Press.000337_to_000360.pfb | 5978 | 17570 | 2.94 |
| Average | 6332.2 | 16308.4 | 2.58 |

Table 5.1: Comparison of execution times of our approach over existing approach and speedup

# CHAPTER 6

# CONCLUSIONS

## 6.1 What have we done so far

We have established that static compilation techniques along with optimization like file I/O fusion can significantly reduce the computation time required for processing large hydrologic data files. We have seen an average speedup of 2.58 times. Also using our approach we could totally eliminate the temporary storage that is being used in the existing implementation.

## 6.2 Future directions

This work has shown using static compilation techniques can significantly enhance the performance of hydrologic data processing systems. This work is a stepping stone towards creating tools that can significantly improve performance for data processing systems without impacting the programmer's productivity.

### 6.2.1 Supporting more hydrologic data processing tools

The hydrologic data processing tools consist of a wide array of functionality. Our library as of now implements only a subset of functionality. We need to implement

more features that enable end users to perform different types of computations related to hydrologic data processing.

# CHAPTER 7

# RELATED WORK

This work builds on research done on various python libraries and domain-specific languages.

## 7.1 Python Libraries

Xarray [7] is an open-source python package that extends the labeled data functionality of Pandas to N-dimensional array-like (tensors) datasets. It provides features for manipulating multi-dimensional datasets, and out-of-core computation to support parallel and streaming computation on larger-than-memory datasets backed by dask [14].

Dask [14] is a flexible library for parallel computing in Python that provides multi-core execution on larger-than-memory datasets and deferred execution. Dask represents the computations in the form of task graphs. The actual computation on these graphs is only done when output is involved. Dask divides arrays into many small pieces, called chunks, each of which is presumed to be small enough to fit into memory. It helps for larger than memory dataset computation.

Dask and Xarray provide various features like parallelism and lazy loading. These features are essential to increase performance and productivity. But, they do not offer features like improving data locality to utilize cache hierarchies. Representing code

in the SPF-IR framework, we can find legal transformations to improve data locality. Hence, we can benefit from cache hierarchies. Also, the proposed library provides differed execution.

## 7.2 Domain Specific Languages/Compilers

Halide [13] is a programming language developed to enhance large-scale image and array processing performance. Halide separates algorithms from schedules. The main advantage is that users can search for many schedules to find the optimal schedule.

PolyMages [10] is also a domain-specific language for the automatic optimization of image processing pipelines. Polymage uses the polyhedral framework for transformations and code generation, providing features like complex fusion, tiling, and storage optimization.

Our approach combines fusing input and output to optimize hydrologic data processing systems. It allows optimally reading data streams, performing computation on them, and immediately writing them back to the disk. It reduces memory footprint and improves the performance of the programs. Halide and Polymage do not consider file input/output in their optimization pipelines.

# REFERENCES

[1] Stephen J Burges. Trends and directions in hydrology. *Water Resources Research*, 22(9S):1S–5S, 1986.

[2] Liz Carolan, Fiona Smith, Vassilis Protonotarios, Ben Schaap, Ellen Broad, Jack Hardinges, and William Gerry. How can we improve agriculture, food and nutrition with open data. *London, UK: Open Data Institute*, 2015.

[3] Chun Chen. Polyhedra scanning revisited. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 499–508, 2012.

[4] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[5] Ralf W Grosse-Kunstleve, Thomas C Terwilliger, Nicholas K Sauter, and Paul D Adams. Automatic fortran to c++ conversion with fable. *Source code for biology and medicine*, 7(1):1–11, 2012.

[6] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[7] Stephan Hoyer and Joe Hamman. xarray: Nd labeled arrays and datasets in python. *Journal of Open Research Software*, 5(1), 2017.

[8] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, University of Maryland at College Park, March 1995.

[9] Benjamin NO Kuffour, Nicholas B Engdahl, Carol S Woodward, Laura E Condon, Stefan Kollet, and Reed M Maxwell. Simulating coupled surface–subsurface flows with parflow v3. 5.0: capabilities, applications, and ongoing development of an open-source, massively parallel, integrated hydrologic model. *Geoscientific Model Development*, 13(3):1373–1397, 2020.

[10] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News*, 43(1):429–443, 2015.

[11] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[12] Tobi Popoola, Ravi Shankar, Anna Rift, Shivani Singh, Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. An object-oriented interface to the sparse polyhedral library. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1825–1831. IEEE, 2021.

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[14] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.

[15] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral optimizations for a data-flow graph language. In *Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers 28*, pages 57–72. Springer, 2016.

[16] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the Sparse Polyhedral Framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7760 LNCS:61–75, 2013.