Automatic Generation of a Real-Time Task Model for Deep Neural Networks

Internship Report by:
Hadi MEKDAD

Academic supervisor:
Prof. Sahar HOTEIT

Laboratory supervisors:
Prof. Mourad DRIDI
Prof. Yasmina ABDEDDAÏM

October 2023

Acknowledgments:

First and foremost, my deepest gratitude goes to my parents and family. Their unwavering support and sacrifices have made it possible for me to embark on this Master's degree journey in Paris. Their resilience and determination have been the driving force behind my success, and I hope to have made them exceedingly proud.

I would also like to express my profound thanks to my friends, both those in Lebanon and those I have had the privilege to meet here in France. They have been my pillars of strength, providing unwavering support and motivation during the most challenging moments of my academic pursuit.

I am immensely grateful to Professors Sahar Hoteit and Mihai Mitrea for their pivotal role in admitting me to this competitive Master's program. Their belief in my potential has been a tremendous source of inspiration and encouragement.

Last but certainly not least, I extend my sincere appreciation to the remarkable team at LIGM, particularly Professors Mourad Dridi and Yasmina Abdeddaim. Their warm welcome, continuous support, and provision of essential resources during my internship have created an ideal environment for me to thrive and accomplish my work.

Each of you has played an indispensable part in my academic journey, and for that, I am deeply thankful.

Table of Contents

Contents

Table of Contents

List of Figures:

List of Tables:

1. Introduction

In an increasingly interconnected world, the demand for systems that can perform tasks in real-time has grown significantly. Real-time systems are those that are required to produce results within strict time constraints, often with response times measured in milliseconds or microseconds. These systems are critical applications across industries such as aerospace, automotive, healthcare, telecommunications, and more. They are designed to meet stringent timing requirements, ensuring that tasks are executed promptly and predictably, thereby guaranteeing the system's reliability and effectiveness.

One of the most interesting fields of real-time systems has been the integration of Deep Neural Networks (DNNs). DNNs are a subset of machine learning algorithms inspired by the human brain's neural structure. They are designed to process and learn from large volumes of complex data, enabling them to recognize patterns, make predictions, and perform classification tasks. While DNNs have gained immense popularity in various applications like image recognition, natural language processing, and recommendation systems, their role in real-time systems is of particular significance.

The integration of DNNs into real-time systems can open up new possibilities for automation, decision-making, and control. However, the incorporation of DNNs into real-time contexts presents a set of challenges. Real-time systems must not only ensure accurate and timely predictions or classifications but also guarantee that these computations are completed within the stipulated time frame. This introduces concerns related to latency, computational efficiency, and resource allocation.

Our project during this internship at LIGM (Laboratoire d'Informatique Gaspard Monge) was to develop a program, which is able to automatically generate a task model from input CUDA files and specification files. By using this model, we will be able to conduct a schedule test to check if it’s schedulable (respects timing constraints of our real time system).

In the following section 2, we will introduce LIGM, the Laboratory where we performed our internship. Following that in section 3, we will talk about the work context and introduce the problem statement. After that in section 4, we will discuss the proposed solution then perform an experiment, show our results and evaluation. Finally, in section 5 we will reflect and conclude on all our work then talk about future works in such domain.

2. LIGM Lab Presentation:

The Laboratory Informatique Gaspard Monge is a computer laboratory, located on the Cité Descartes campus in Champs s/ Marne, in 3 buildings (ESIEE, Bâtiment Copernic and Bâtiment Coriolis).

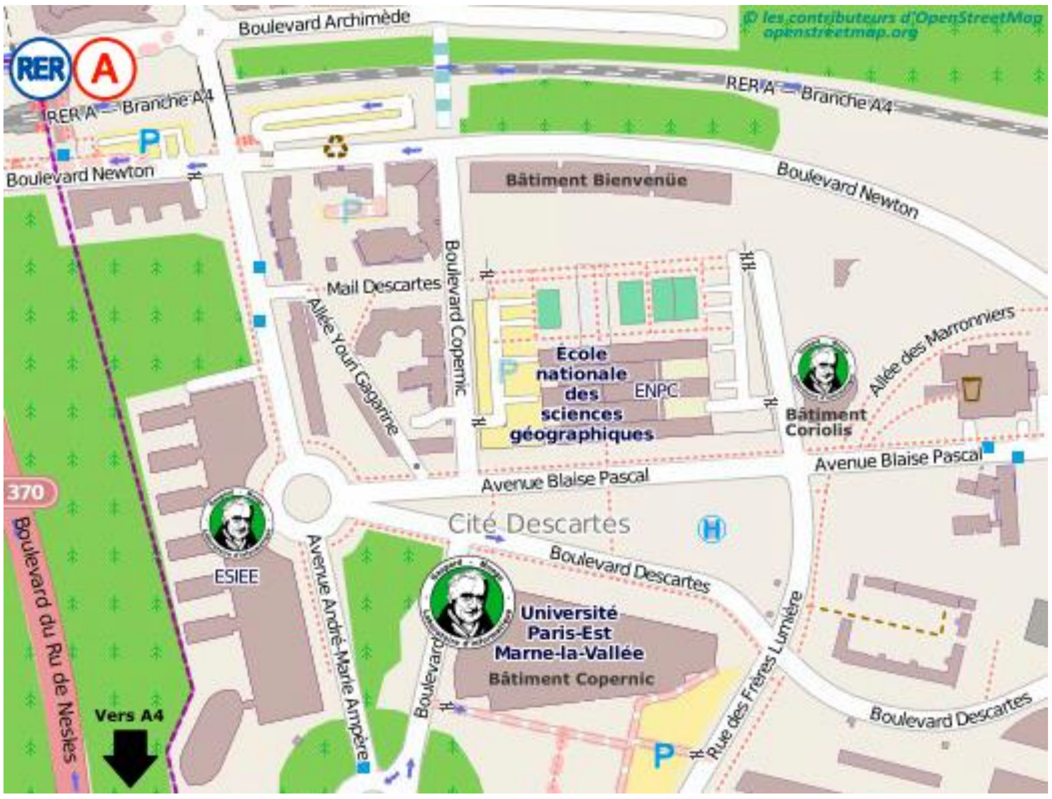


Figure 1-LIGM Plan.

There are 6 research teams, for a total of 80 permanent and 50 non-permanent researchers. These teams are A3SI (Algorithms, architectures, image analysis and synthesis), ADA (Discrete algorithms and applications), BAAM (Database, automaton, analysis of algorithms and models), COMBI (Algebraic combinatorics and symbolic computation), LRT (Software, networks and real time), and SIGNAL (Signal and communication). Members of LIGM are involved in computer science courses at the Gaspard-Monge Institute, the IUT of Marne-la-Vallée, ESIEE Paris and the Ecole des Ponts ParisTech.

My team and I are a part of the LRT team working in the ESIEE building specializing in the domain of real-time systems. My supervisors Prof. Mourad DRIDI & Prof. Yasmina ABEDDAIM are a part of this team. My internship took place in ESIEE part of the LIGM lab.

3. Context & Problem Statement:

The context of our work relies in real-time systems, GPU, and real-time DNNs. We will also use the notion of CUDA in GPUs while implementing DNNs. Below are some definitions of what we work with.

3.1 Real-time Systems:

3.1.1 Definition:

A real-time system is a computing system that must process and produce results within predetermined time limits. (Insup Lee, 2007)

Real-time systems are widely used in various domains where timely and reliable processing is critical such as autonomous vehicles, surveillance and security, patient monitoring, flight control, etc.

3.1.2 Types:

There are several types of real-time systems depending on the criticality of timing constraints such as hard real-time systems and soft real-time systems.

Hard Real-Time Systems:

Hard real-time systems have the most stringent timing requirements. In these systems, missing a deadline is considered catastrophic and unacceptable. The system must guarantee that tasks are completed within their specified time bounds, even under worst-case conditions. Failure to meet a deadline can lead to system failures, safety hazards, or significant financial losses. (Neil Audsley, 1995) (Giorgio Buttazzo, 2005) (Layland, 1973)

Examples:

- Air traffic control systems: Ensuring the safe separation of aircraft requires real-time processing of radar data and communication updates.
- Medical devices: Life-saving equipment like pacemakers or infusion pumps must respond to critical patient conditions without delay.
- Automotive safety systems: Airbag deployment and collision avoidance systems require immediate and precise actions to ensure passenger safety.

Soft Real-Time Systems:

Soft real-time systems have less stringent timing requirements compared to hard real-time systems. While meeting deadlines is still important, occasional misses may be tolerated without causing catastrophic failures. (Neil Audsley, 1995) (Giorgio Buttazzo, 2005) (Layland, 1973)

Examples:

- Multimedia streaming: Video conferencing and online streaming aim to provide a seamless user experience, where minor delays may not be critical.
- Online gaming: While low latency is crucial for a responsive gaming experience, occasional delays might not result in significant consequences.
- Online transaction processing: E-commerce systems aim to process transactions quickly, but minor variations in response times may be acceptable.

Mixed Real-Time Systems:

Mixed real-time systems, also known as hybrid real-time systems, combine elements of both hard real-time and soft real-time characteristics. These systems handle a mix of tasks with varying levels of timing requirements, allowing for a more flexible and balanced approach to meeting deadlines. Mixed real-time systems are particularly useful when some tasks are critical and must meet strict timing constraints, while others can tolerate occasional delays without catastrophic consequences.

In mixed real-time systems, tasks are categorized based on their criticality and timing requirements. Critical tasks, which cannot miss their deadlines without severe consequences, are treated as hard real-time tasks. Non-critical tasks, which can tolerate some degree of delay, are treated as soft real-time tasks. The system's design and scheduling algorithms then aim to prioritize critical tasks while still ensuring satisfactory performance for non-critical tasks.

Examples:

- Smart Home Systems: They might have critical tasks like fire or security alarms that require immediate response for safety. Less critical tasks, such as adjusting room lighting or managing thermostats, can tolerate minor delays without compromising overall functionality.
- Health Monitoring Devices: Wearable health devices prioritize real-time alerts for critical conditions like abnormal heart rhythms. Less time-sensitive tasks, such as tracking daily activity levels, can have some flexibility in timing.

3.1.3 Real-Time Scheduling:

In real-time scheduling, a real-time system is modeled into a set of real-time tasks to be executed in a given architecture (CPU, GPU, etc.) within strict timing constraints. (N. Audsley)

A task is a unit of work that needs to be executed by a computer system. Tasks in real-time scheduling often have associated deadlines, priorities, and execution times. They represent the individual processes or activities that the system must handle.

There are some important notions to define a real-time task which are:

- **Period:** The period of a task refers to the time interval between successive releases or activations of the task. In periodic real-time systems, tasks are executed at regular intervals, and the period defines how often a task repeats its execution.
- **Deadline:** A deadline is the time by which a task must complete its execution. It is a critical parameter in real-time systems, as missing a deadline can lead to system failures or degraded performance. Deadlines are often categorized into hard deadlines (must be met without exception) and soft deadlines (can be occasionally missed, but with some performance degradation).
- **Priority:** Priority is used to determine the order in which tasks are executed when multiple tasks are ready to run simultaneously. Tasks with higher priority values are executed before those with lower priorities. Priority assignment is crucial for meeting the timing requirements of high-priority tasks while ensuring that lower-priority tasks do not cause unacceptable delays.

There are some types of Real-Time scheduling among of them are:

- **Preemptive Scheduling:** In preemptive scheduling, higher-priority tasks can interrupt the execution of lower-priority tasks. This ensures that the most critical tasks are executed promptly when they become eligible to run.
- **Non-preemptive Scheduling:** In non-preemptive scheduling, once a task begins execution, it is not interrupted until it completes or voluntarily yields the CPU/GPU. This can be suitable for certain applications but may not be ideal for real-time systems with strict deadlines.

3.2 DNN:

3.2.1 Definition:

Deep Neural Networks (DNNs) are networks which leverage the biological structure of the brain to form a similar one following how neurons are connected in the brain. They use sophisticated mathematical modeling to process data in complex ways.

DNNs are consisted of neurons. They are like computational nodes/units where they receive an input, perform an operation, and give an output.

Neurons grouped together form what is called a layer. Layers can be connected to each other by connecting their respective neurons. A deep neural network has a certain level of complexity and it also contains more than two layers. (Dehua Zheng, January, 2021) (Ranjeet Kaur, 2022)

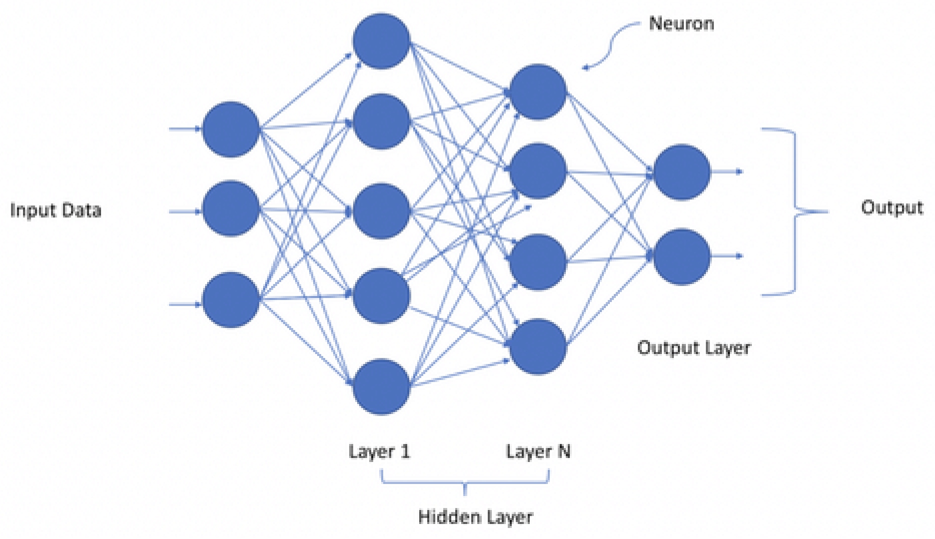


Figure 2-Illustration of a DNN.

The first layer in a DNN receives an input, it processes this input through its neurons and then outputs the result to the next layer. The next layer receives the output of the preceding layer as its input and processes it, and so on... The final layer usually outputs the overall output of the network.

Deep Neural Networks (DNNs) have two phases; a training phase which involves optimizing the network's parameters using labeled data to learn patterns & relationships, and an inference phase which applies the trained network to make predictions or classifications on new unseen data without further parameter adjustments. In our work, we are interested in the inference phase.

3.2.2 Real-Time DNNs:

Real-time DNNs are DNNs which are executed on a real-time system, having a timing constraint.

Some of these Real-Time DNNs are YOLO versions 1-8/tiny, ResNet-18/50/101 etc. These DNNs are mostly responsible for object detection and classification which is related to our work for the context of autonomous driving. In our work, we used YOLO v3-tiny and ResNET-18.

3.3 Graphics Processing Unit (GPU):

3.3.1 Definition:

A Graphics Processing Unit (GPU) is a specialized hardware component designed to accelerate the processing of graphics-related tasks. Originally developed for rendering images and videos in computer graphics, GPUs have evolved into highly parallel processors capable of handling a wide range of computationally intensive tasks beyond graphics. They excel at performing repetitive calculations simultaneously, making them well-suited for tasks such as scientific simulations, machine learning, and other parallel computing applications. (Jamshed, December, 2015)

Nvidia GPUs, known for their advanced CUDA (Compute Unified Device Architecture) architecture, are specialized hardware optimized for parallel processing tasks. Originally developed for graphics, they've become essential for diverse applications, including scientific simulations and machine learning, thanks to their ability to execute multiple calculations in parallel. In our work, we will be focusing on the notion of CUDA alongside Nvidia GPUs.

3.3.2 GPU Architecture & CUDA:

The GPU consists of multiple crucial elements. Some of these main parts are:

- **Streaming Multiprocessors (SMs):** A Streaming Multiprocessor (SM) is the fundamental processing unit within a GPU. It consists of multiple processing cores and various components necessary for executing instructions in parallel. Each SM can perform independent computations and manage its own resources, such as registers and memory caches. The number of SMs in a GPU determines its overall processing power and parallel computing capabilities.
- **Kernels:** They are functions that are designed to execute in parallel across multiple threads on the GPU. Kernels are the primary units of work in GPU computing, and they define the tasks to be performed on individual data elements.
- **Blocks:** A block, also known as a thread block, is a group of threads that are scheduled to execute together on an SM. Threads within a block can communicate and synchronize with each other using shared memory. The size of a block is determined by the programmer and can vary based on the specific application.
- **Threads:** Threads are individual units of execution within a block. Each thread executes the same kernel code but operates on different data elements. Threads within a block can share data through shared memory and collaborate on specific tasks.
- **Grids:** A grid is a collection of thread blocks that collectively execute a kernel. Grids provide a higher-level organization of parallel execution, allowing multiple blocks to work together on a problem. Grids are often used to process large datasets in parallel by distributing the work among different thread blocks.
- **Stream:** It is a sequence of tasks or commands that are submitted for execution and processed in a specific order. These tasks typically include operations related to graphics rendering, general-purpose computing, or data processing. The concept of streams is fundamental to achieving parallelism and efficient utilization of GPU resources.

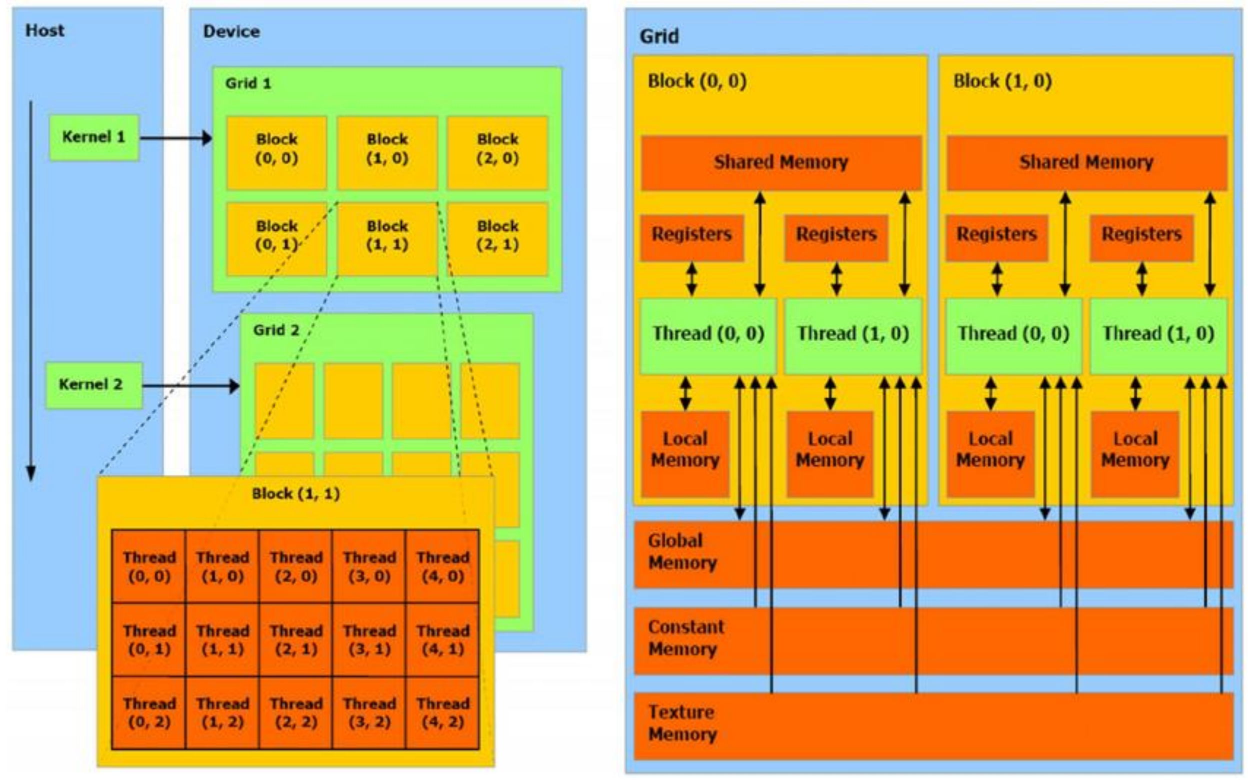


Figure 3-GPU Architecture showing Kernels, Grids, Blocks, and Threads. *(Morrison, 2023)*

3.4 Problem Statement:

Deep Neural Networks have proven to be highly effective in tasks such as image recognition, natural language processing, and complex pattern recognition. However, these networks are computationally intensive and often require substantial processing time to produce results. To mitigate this, DNNs are commonly executed on hybrid architectures that combine Central Processing Units (CPUs) and Graphics Processing Units (GPUs). GPUs excel at parallel processing and can significantly accelerate the inference process of DNNs.

The primary concern arises when integrating DNNs into real-time systems or applications where meeting strict timing constraints is crucial. Classical real-time task models are designed to analyze the timing behavior of traditional software tasks with predictable execution times. These models do not account for the intricacies of DNN execution on heterogeneous CPU-GPU architectures, which introduces complexities that cannot be accurately captured by conventional real-time analysis techniques.

Take the case “Memory Management” as an example where efficient data transfer between the CPU and GPU is essential for DNN execution. However, data transfer times can be non-deterministic, affecting the overall timing behavior of the system.

In our work, we address this problem by proposing an automatic generation of a task model that is able to take into account the characteristics of the embedded DNN and the real-time system. This model is able to undergo a scheduling analysis to see if it’s suitable for the corresponding real-time system.

3.5 Related Works:

In (Weiguang Pang, 2023), the authors address the challenge of efficiently deploying multiple real-time Deep Neural Network (DNN) tasks with varying timing requirements on a single embedded GPU. They focus on CUDA streams with priority & multiple DNN tasks with real-time constraints deployed on a heterogeneous CPU–GPU embedded platform. They also introduce a task model similar to our work using basic scheduling rules. They propose a GPU CUDA stream priority assignment strategy to meet the real-time requirements of multi-DNN tasks while maximizing GPU resource utilization.

In (An Zou, 2023), the authors propose a RTGPU, a model which achieves high schedulability to meet the stringent time constraints of real-time. It efficiently schedules multiple GPU applications while providing rigorous real-time guarantees.

However, in our approach we also talk about the usage of a single CUDA stream as well as multiple CUDA streams. We underline the advantages and disadvantages for each case. We also introduce a set of rules yielding to the automatic generation of a task model with its respective DNN parameters. Our program yields to a generation of a *unique* task model for every case, unlike using the same task model for all cases.

4. Proposed Solution:

We propose a task model for a real-time DNN deployed over a GPU architecture. This model is a set of tasks which describe simultaneously the characteristics of a real-time DNN and a GPU architecture.
In the following sub-section, we will give an overview of this task-model.

4.1 Task Model:

In a GPU architecture, kernels can be executed over a CPU part or a GPU part.
Kernels over GPU can be executed for Image processing as an example.
Kernels over CPU can be executed for data transfer from Host (CPU) to Device (GPU) or vice versa as an example.

For this reason, we consider GPU tasks and CPU tasks in our proposed model.

We model these tasks as two sets of tasks:

- CPU Task Set: describes tasks executed on CPU: Ω_{CPU}
- GPU Task Set: describes tasks executed on GPU: Ω_{GPU}

Each task set is composed of *n* tasks.

$\Omega_{CPU} = \{\zeta_1, \zeta_2, \zeta_3, \dots, \zeta_n\}$
$\Omega_{GPU} = \{\zeta_1, \zeta_2, \zeta_3, \dots, \zeta_n\}$

Each task ζ_i of the CPU Task Set Ω_{CPU} is characterized by the following parameters:

$\zeta_i = \{R_i, P_i, D_i, Pr_i, C_i, No_i, Ne_i\}$

- ***R_i* - Release Time:** It is point in time when the task becomes available for execution. (Pathshala)
- ***P_i* – Period:** It is a time interval between consecutive releases or instances of execution of the same task.
- ***D_i* - Implicit Deadline:** It is a time limit by which a task should be completed.
- ***Pr_i* - Fixed Priority:** It is assigned to a task in order to determine its order of execution. A fixed priority remains constant throughout the execution process.
- ***C_i* - Computation Time:** We use the definition of Worst-Case Execution Time (WCET)*. consider a measurement-based WCET estimation.
- ***No_i* - Node:** Identifies the CPU running the task, i.e. to which CPU each task is assigned to.
- ***Ne_i* - Next:** is the precedence constraint function which determines the following task to the current one.

Each task ζ_j of the GPU Task Set Ω_{GPU} is characterized by the following parameters:

$\zeta_j = \{R_j, P_j, D_j, Pr_j, C_j, S_j, T_j, Da_j, Si_j, Ne_j\}$

- ***R_j* - Release Time:** The same definition in the CPU Task Set.
- ***P_j* – Period:** The same definition in the CPU Task Set.
- ***D_j* - Implicit Deadline:** The same definition in the CPU Task Set.
- ***Pr_j* - Fixed Priority:** The same definition in the CPU Task Set.
- ***C_j* - Computation Time:** It's also specified as WCET. The value of this parameter is also dependent on the quality of the data being handled.
- ***S_j* - Stream:** specifies to which stream is the task assigned, as we can have multiple streams in our model.
- ***T_j* - Type:** specifies the type of operation the task performs. E.g. addition, convolution, etc.
- ***Da_j* - Data:** specifies the data that is used by the task.
- ***Si_j* - Size:** specifies the number of sub-tasks (blocks) of the task.
- ***Ne_i* - Next:** The same definition in the CPU Task Set.

* *Worst-Case Execution Time (WCET)*: The WCET of a task is the longest execution time of this task. (Reinhard Wilhelm, 2008)

We propose this model as it is able to capture the characteristics of a real time DNN as well as the GPU architectures. It compacts them in parameters distributed among two task sets; The CPU Task Set and the GPU Task Set. Furthermore, the task model accurately captures the parameters indicating all the necessary information we need to gather in order to perform a schedulability analysis of our system.

4.2 Generation of Task Model:

Overview:

In this work, we propose an automatic generation of the task model. From source files, we are able to generate this model.

The used source files are:

1. .cu file (CUDA file)
2. Specification file (.txt file)

The produced file is a .xml file which contains the task model as shown in the figure below:

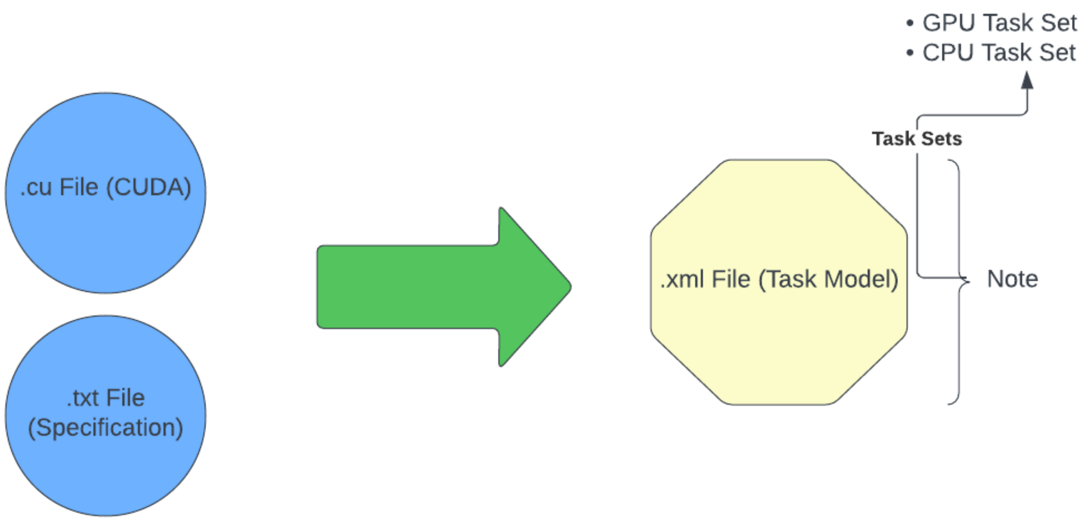


Figure 4–Task Model Generation.

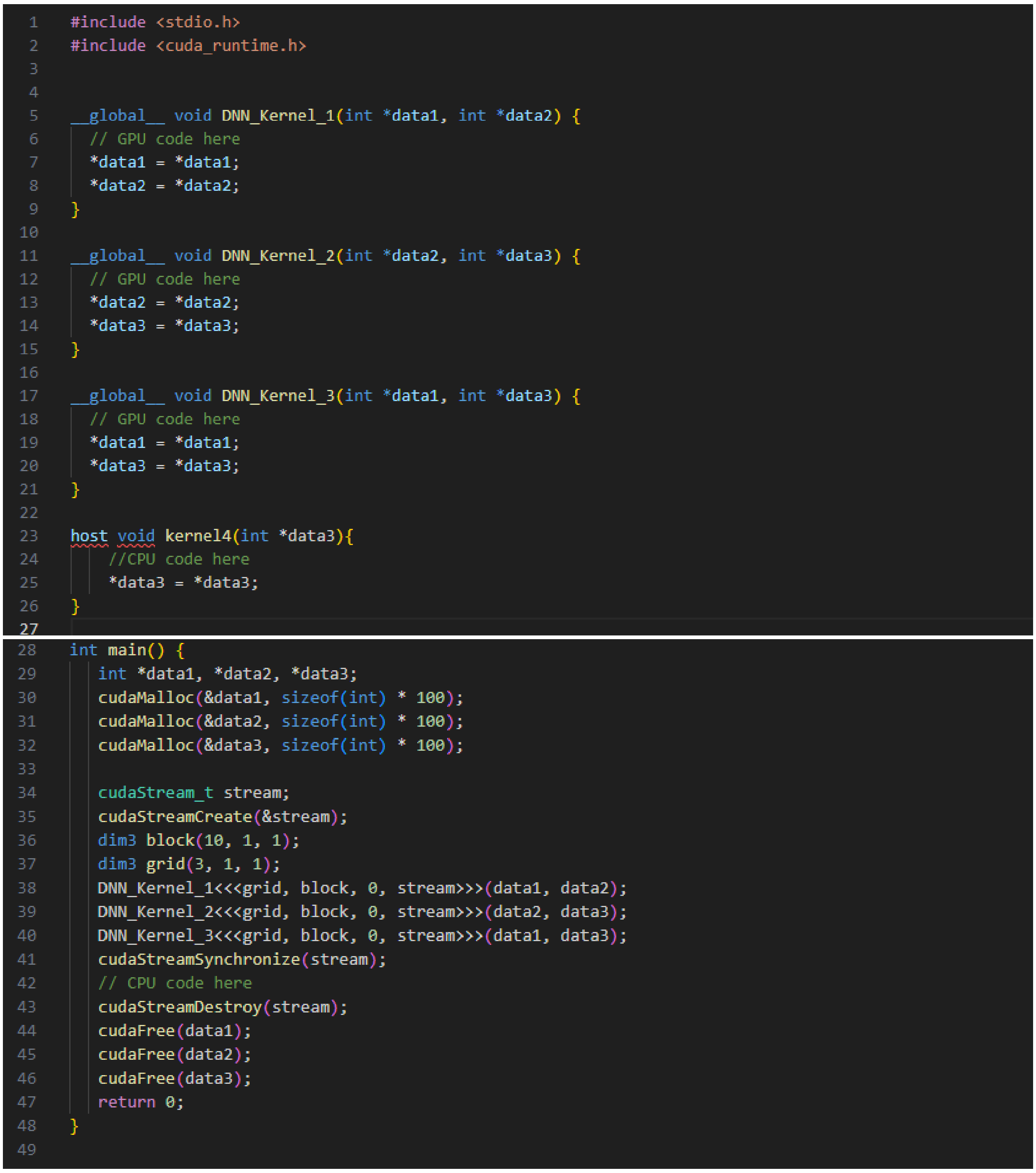
CUDA File:

In order to deploy a DNN over a Nvidia GPU, a CUDA source file containing the DNN is executed on the GPU to run the DNN. As a result, CUDA files contain parameters of the DNN and hence are an essential ingredient for generating our task model.

The CUDA file contains the parameters of the DNN and a specification file contains the parameters of a real-time system.

Example:

Below is an example of a CUDA code for one DNN inference application with one CPU task. The DNN application is composed of 3 GPU kernels which use one stream. Kernel 1 named DNN_Kernel_1 uses data1 and data2. Kernel 2 named DNN_Kernel_2 uses data2 and data3. Kernel 3 named DNN_Kernel_3 uses data1 and data3.



```
#include <stdio.h>
#include <cuda_runtime.h>


__global__ void DNN_Kernel_1(int *data1, int *data2) {
  // GPU code here
  *data1 = *data1;
  *data2 = *data2;
}

__global__ void DNN_Kernel_2(int *data2, int *data3) {
  // GPU code here
  *data2 = *data2;
  *data3 = *data3;
}

__global__ void DNN_Kernel_3(int *data1, int *data3) {
  // GPU code here
  *data1 = *data1;
  *data3 = *data3;
}

host void kernel4(int *data3){
    //CPU code here
    *data3 = *data3;
}

int main() {
   int *data1, *data2, *data3;
   cudaMalloc(&data1, sizeof(int) * 100);
   cudaMalloc(&data2, sizeof(int) * 100);
   cudaMalloc(&data3, sizeof(int) * 100);

   cudaStream_t stream;
   cudaStreamCreate(&stream);
   dim3 block(10, 1, 1);
   dim3 grid(3, 1, 1);
   DNN_Kernel_1<<<grid, block, 0, stream>>>(data1, data2);
   DNN_Kernel_2<<<grid, block, 0, stream>>>(data2, data3);
   DNN_Kernel_3<<<grid, block, 0, stream>>>(data1, data3);
   cudaStreamSynchronize(stream);
   // CPU code here
   cudaStreamDestroy(stream);
   cudaFree(data1);
   cudaFree(data2);
   cudaFree(data3);
   return 0;
}
```

Figure 5-CUDA code with 3 GPU kernels and 1 CPU function.

Specification File:

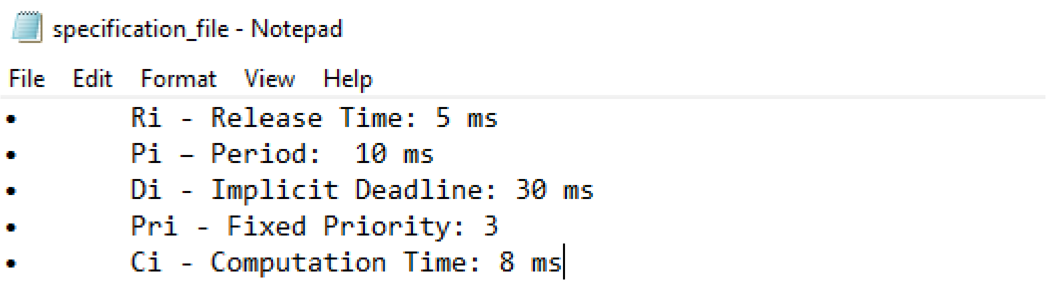
The specification file (.txt File) contains some of the parameters which describe our real-time system and GPU architecture. Some of these parameters are:

- *Release Time*
- *Period*
- *Implicit Deadline*
- *Fixed Priority*
- *Computation Time*

These parameters are simply written in a .txt file.

Example:

Below is an example of a specification file (.txt File) which contains some of the parameters of the task model:



```
•       Ri - Release Time: 5 ms
•       Pi – Period:  10 ms
•       Di - Implicit Deadline: 30 ms
•       Pri - Fixed Priority: 3
•       Ci - Computation Time: 8 ms
```

Figure 6-Specification File.

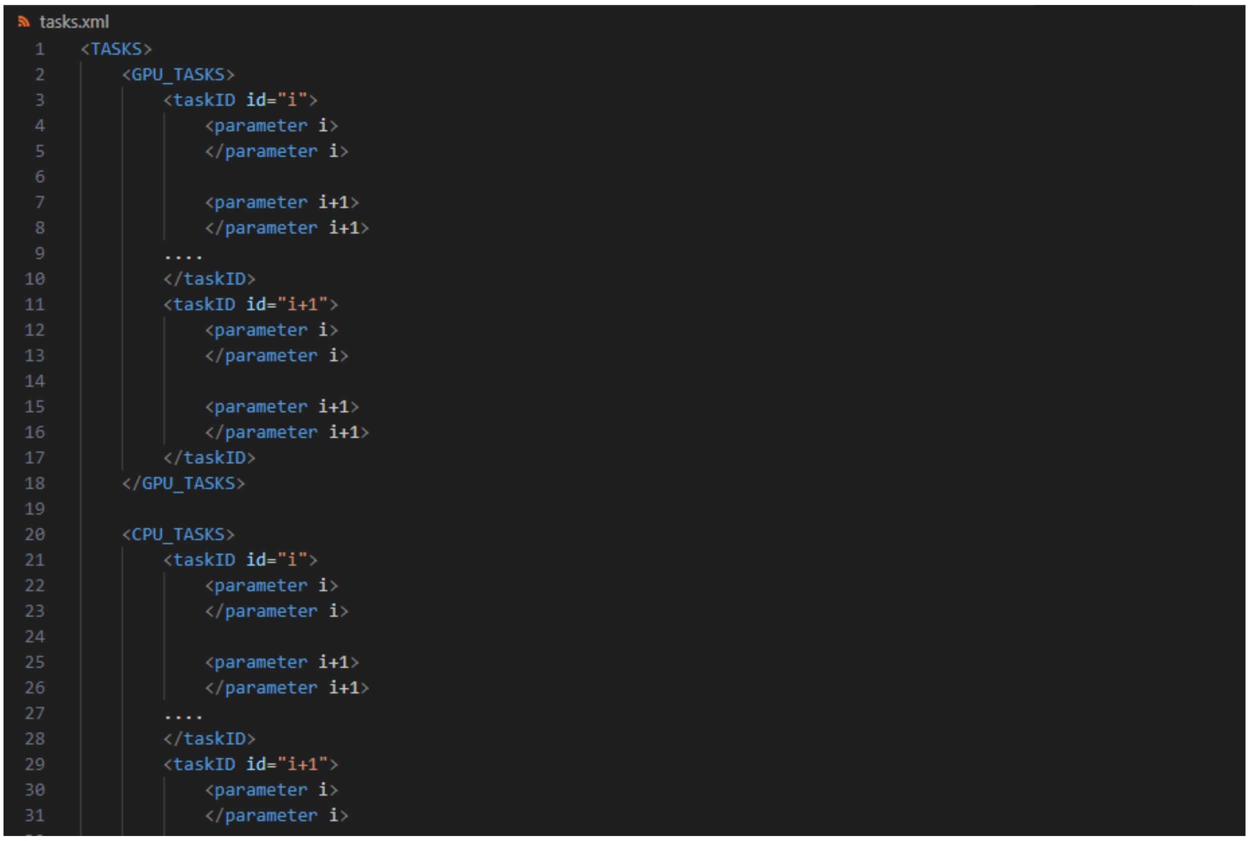
XML File:

The .xml file is the file which contains the generated task model. It contains both CPU and GPU Task Sets Ω_{CPU} and Ω_{GPU}.

We generate the task model in a .xml file format in order to use this file later on for a tool which conducts a schedulability test for our model. We can consider then that the .xml file is the backbone of our scheduling analysis test. More explanation about this tool is mentioned in the following section.

Example:

Below is an example of a .xml file which describes the form of the file:



tasks.xml

```
<TASKS>
    <GPU_TASKS>
        <taskID id="i">
            <parameter i>
            </parameter i>

            <parameter i+1>
            </parameter i+1>
        ....
        </taskID>
        <taskID id="i+1">
            <parameter i>
            </parameter i>

            <parameter i+1>
            </parameter i+1>
        </taskID>
    </GPU_TASKS>

    <CPU_TASKS>
        <taskID id="i">
            <parameter i>
            </parameter i>

            <parameter i+1>
            </parameter i+1>
        ....
        </taskID>
        <taskID id="i+1">
            <parameter i>
            </parameter i>
```

Figure 7-.xml File.

4.3 Implementation:

In order to collect the parameters into task sets, we first propose a set of rules to detect the parameters we need in the CUDA file.

Rules:

Before we begin implementing our algorithm for generation, we propose rules to be applied in order to accurately capture the parameters we need.

1. **Release time, period, deadline, priority, & Worst-Case Execution Time (WCET)** can be identified through our specification file.

2. Each CUDA kernel function that its name begins with “*global*” is a **GPU** kernel. While “*host*” indicates that the kernel is that of a **CPU** one.

3. Due to memory management, we will associate for each GPU CUDA kernel a CPU task as well. Meaning we will create two tasks for every GPU function, one that of a GPU to do the desired computation, and one that of a CPU that is responsible for memory management. (Copying data from CPU to GPU and the vice versa).

4. When **stream** variable is not indicated or defined, we will assume that there is only one stream in our code. Otherwise, "*CUDAStream_t*" function defines the stream variables and hence their number indicates the number of our streams in the code. We can also identify the stream parameter through the arguments of the calling of the kernel function.

5. We can identify **Next** through the order of our kernels within a stream, and since we assume dependencies, the next kernel to execute is just the following kernel function in the queue of the its stream.

6. We can identify our **data** through the arguments of calling our kernel function.

7. The **type** of the operation performed can be identified through the name of the kernel function.

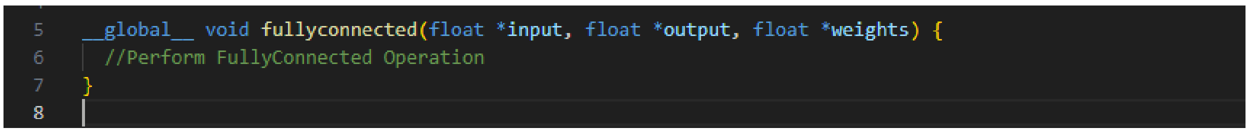
8. The **size** is the number of our blocks which can be found through the function dim3 gridDim ($x_0, x_1, \dots, x_i, \dots, x_n$). Size can be determined through the dimensions specified in the code by multiplying the dimension as follows: size = (dimension[0])*(dimension[1])…. = $\sum_0^n x_i$

9. The **Node** parameter can be known through the name associated to kernels.

Clarifying Examples:

Example 1:

Rule 2 can be applied here. We can deduce from "__global__" that the following kernel represents a GPU one.


```
__global__ void fullyconnected(float *input, float *output, float *weights) {
  //Perform FullyConnected Operation
}
```
Figure 8-Rule 2 Illustration.

Example 2:

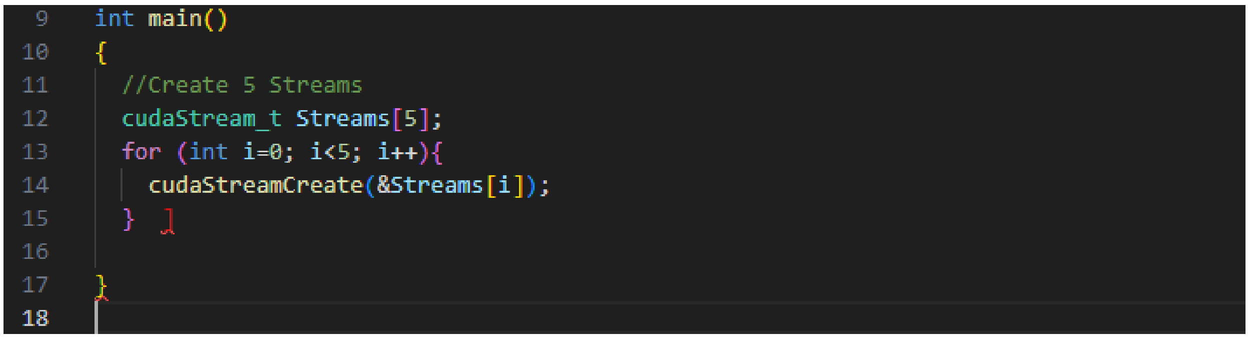
Rule 4 elaborates the creation of streams and how to spot which stream we are in when calling a specific kernel.

Spotting the stream:

*DNN_Kernel1<<<grid, block, 0, **streamA**>>>(data1, data2);*

Associated Stream: **StreamA**

Creating Streams:



```
int main()
{
    //Create 5 Streams
    cudaStream_t Streams[5];
    for (int i=0; i<5; i++){
        cudaStreamCreate(&Streams[i]);
    } ]

}
```

Figure 9-Creating Streams.

Example 3:

Rule 6 extracts used data from the calling our kernel:

*DNN_Kernel1<<<grid, block, 0, streamA>>>(**data1**, **data2**);*
Data: **data1** and **data2.**

Example 4:

Rule 7 extracts type of operation performed from the kernel name:

*__global__ void **fullyConnected**(float* input, float* output, float* weights)*
{
//code of fully connected function
}

Type: **Fully Connected**

Example 5:

Rule 8 states that the size can be calculated from dim3 gridDim API:

Dim3 gridDim**(2,1)**

Size = (2)*(1) = **2 blocks**

Example 6:

Rule 9 states that node can be indicated through name of kernel function:

*__global__ void **residualBlock_CPU1**(float* input, float* output, float* weights)*
{
// Perform the residual block operation
}

Node: **CPU1**

Algorithm:

After proposing our rules, we develop a program which applies these rules in order to capture the parameters and produce the task model. We call it the DTM Generator.

It takes as input a .cu file (CUDA file) and outputs an .xml file containing CPU & GPU task sets which describe the parameters of our model.

We choose the programming language as python for the easy access to libraries which can read and write files in loops. We write the parameters of our task model in the .xml output file as this file can be later used as input for a scheduling tool called **cheddar*.

** Cheddar:* is a real-time scheduling tool/simulator. Cheddar allows us to model software architectures of real-time systems and check their schedulability or other performance criteria. As many schedulability analysis tools, schedulability can be assessed by scheduling simulations or feasibility tests.

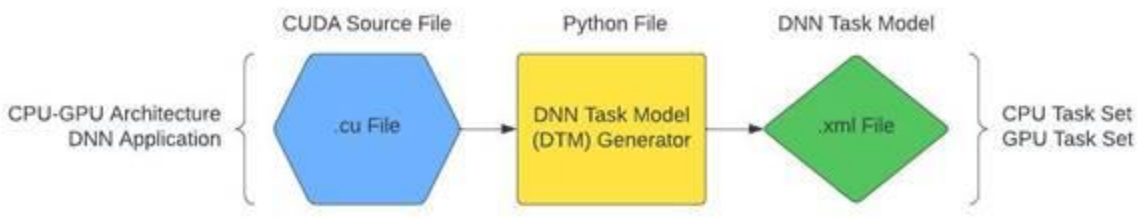


Figure 10 -Illustration of the Task Model Generation.

In short, the idea is to produce an .xml output file from our CUDA input file. For that we propose the following steps our algorithm will perform to produce the model we need.

- **I Reading our input files.**
- **II Scanning our files for keywords.**
- **III Applying the specific rule to each associated keyword.**
- **IV Deduce the specific parameter of our task model.**
- **V Write the parameter in the .xml file.**

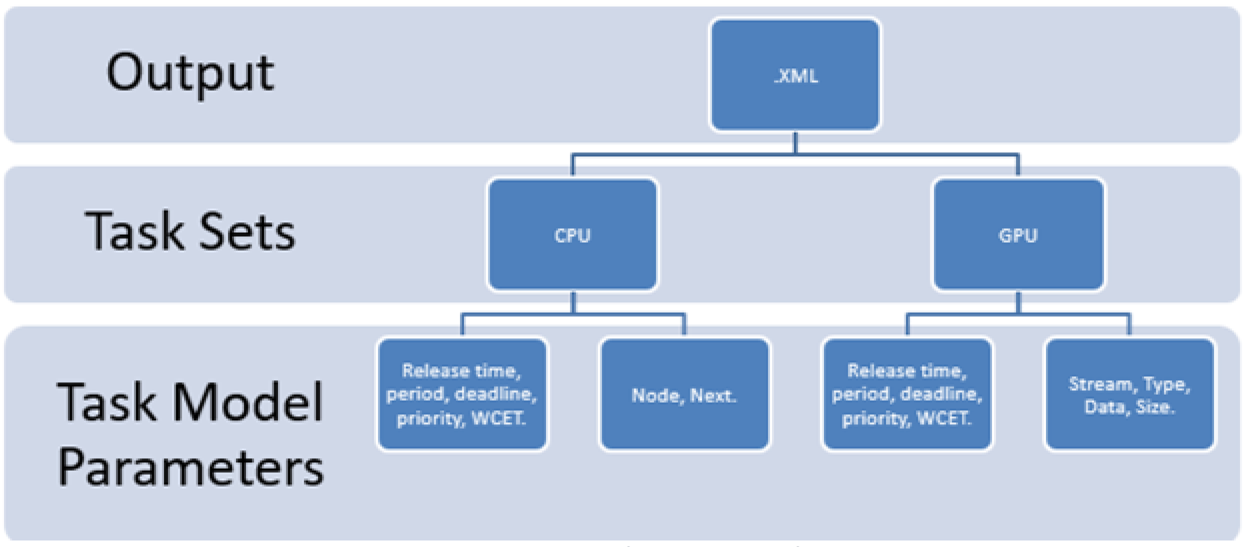
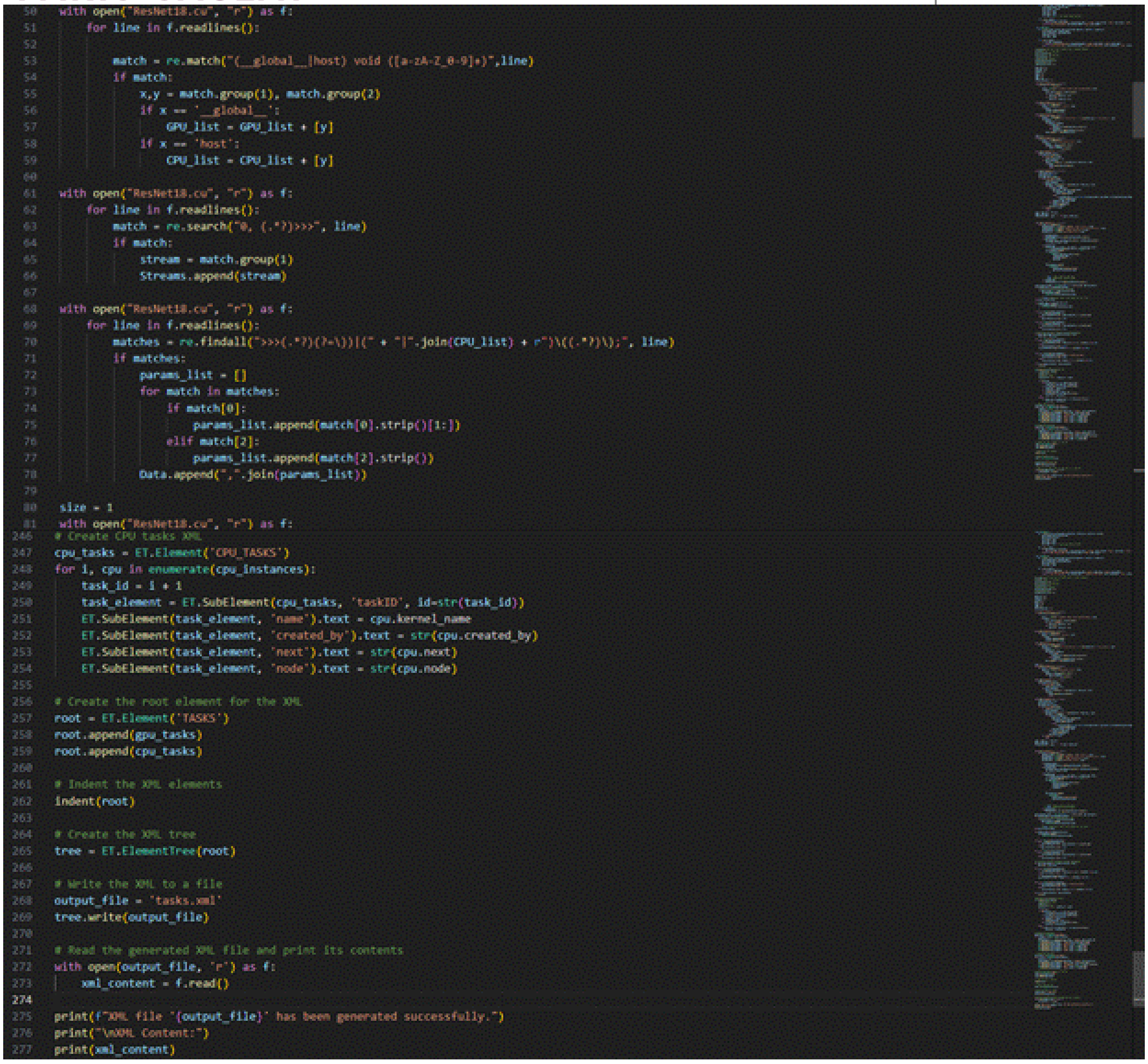

Output
.XML
Task Sets
CPU
GPU
Task Model Parameters
Release time, period, deadline, priority, WCET.
Node, Next.
Release time, period, deadline, priority, WCET.
Stream, Type, Data, Size.

Figure 11-Contents of the .xml output file.

4.4 Implementation details:

Our DTM Generator program is written in python. It takes a .cu input file and produces the .xml output file based on performing reading & scanning the input file for keywords that were stated in the *Rules* section. Our python script of the DTM Generator reads a CUDA source code file and extracts information about GPU and CPU tasks defined in the code. It then generates an .xml file containing the extracted parameters. Here's a brief overview of what the script does:

1. It defines two classes, *GPUElement* and *CPUElement*, to represent GPU and CPU tasks, respectively.
2. It initializes various lists and dictionaries to store information about GPU and CPU tasks, as well as other relevant data.
3. It reads the input file multiple times to extract information such as GPU and CPU kernel names, stream names, data passed to kernels, and other details.
4. It creates instances of *GPUElement* and *CPUElement* to represent GPU and CPU tasks based on the extracted information.
5. It establishes relationships between tasks by setting the Next attribute for each task according to Rule 5.
6. It generates XML content that represents the extracted task information, including kernel names, streams, data, and other attributes.
7. It writes the XML content to an output file named "tasks.xml."
8. Finally, it prints the generated XML content and a confirmation message.



```python
with open("ResNet18.cu", "r") as f:
    for line in f.readlines():

        match = re.match("(__global__|host) void ([a-zA-Z_0-9]+)",line)
        if match:
            x,y = match.group(1), match.group(2)
            if x == '__global__':
                GPU_list = GPU_list + [y]
            if x == 'host':
                CPU_list = CPU_list + [y]

with open("ResNet18.cu", "r") as f:
    for line in f.readlines():
        match = re.search("0, (.*?)>>>", line)
        if match:
            stream = match.group(1)
            Streams.append(stream)

with open("ResNet18.cu", "r") as f:
    for line in f.readlines():
        matches = re.findall(">>>(.*?)(?=\))|(" + "|".join(CPU_list) + r")\((.*?)\);", line)
        if matches:
            params_list = []
            for match in matches:
                if match[0]:
                    params_list.append(match[0].strip()[1:])
                elif match[2]:
                    params_list.append(match[2].strip())
            Data.append(",".join(params_list))

size = 1
with open("ResNet18.cu", "r") as f:
# Create CPU tasks XML
cpu_tasks = ET.Element('CPU_TASKS')
for i, cpu in enumerate(cpu_instances):
    task_id = i + 1
    task_element = ET.SubElement(cpu_tasks, 'taskID', id=str(task_id))
    ET.SubElement(task_element, 'name').text = cpu.kernel_name
    ET.SubElement(task_element, 'created_by').text = str(cpu.created_by)
    ET.SubElement(task_element, 'next').text = str(cpu.next)
    ET.SubElement(task_element, 'node').text = str(cpu.node)

# Create the root element for the XML
root = ET.Element('TASKS')
root.append(gpu_tasks)
root.append(cpu_tasks)

# Indent the XML elements
indent(root)

# Create the XML tree
tree = ET.ElementTree(root)

# Write the XML to a file
output_file = 'tasks.xml'
tree.write(output_file)

# Read the generated XML file and print its contents
with open(output_file, 'r') as f:
    xml_content = f.read()

print(f"XML file '{output_file}' has been generated successfully.")
print("\nXML Content:")
print(xml_content)
```

Figure 12-DTM Generator

The output of our generator is a .xml file containing our Task Model depending on the CUDA input file, each Task Model will be unique in regards to the situation.

4.5 Experiment:

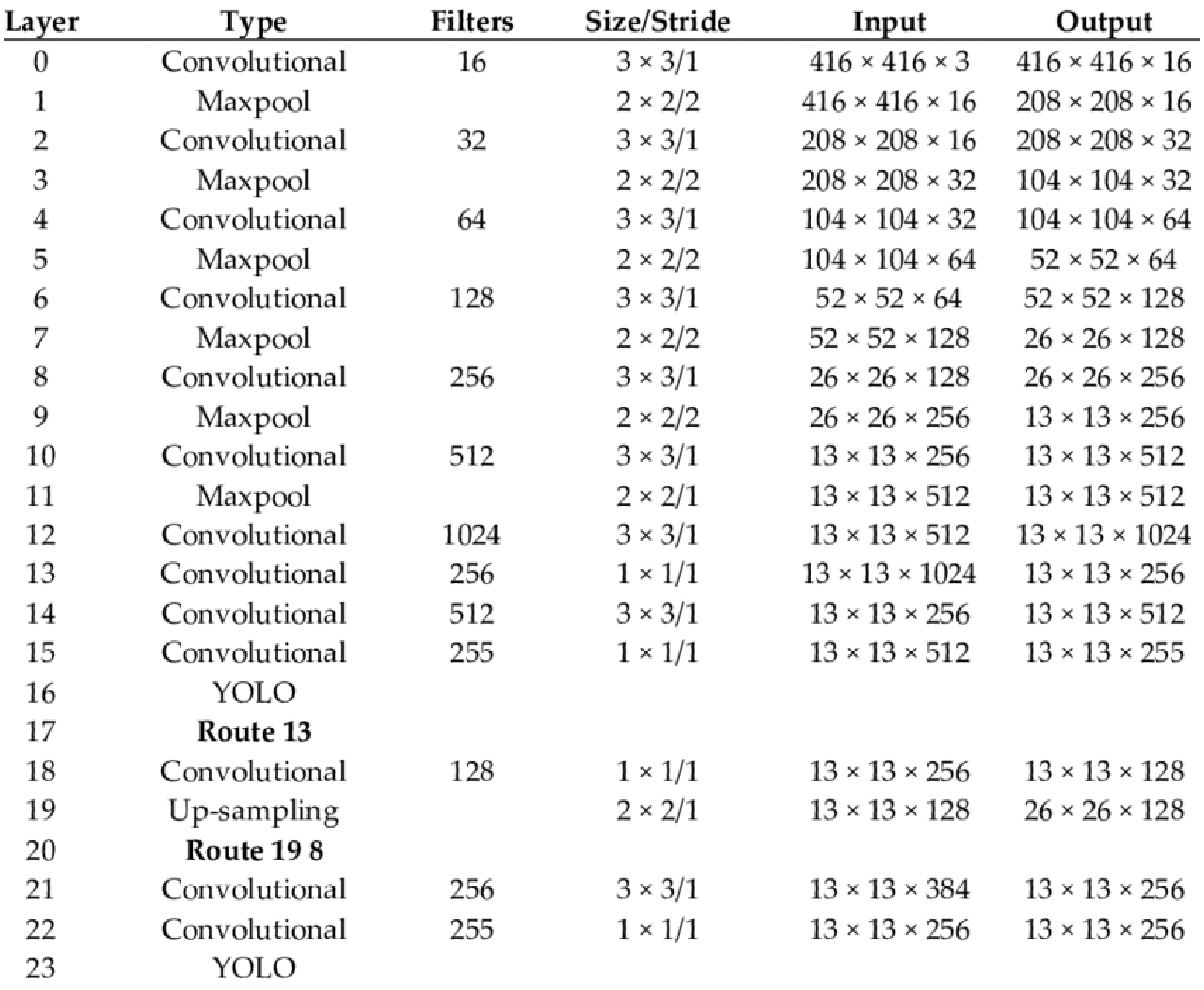
We used YOLO v3 tiny and ResNet-18 DNNs for an experiment to verify our algorithm and generate the task model for each case, we first start by building CUDA files of YOLO v3-tiny and ResNet-18 which respect their official network architecture.

YOLO v3-tiny:

YOLO is a model generally known for object detection purposes. It can identify and locate objects in images and video streams. YOLO has been improved throughout the years, hence, there are many versions of YOLO this day. In our work, we chose to use YOLO v3-tiny. YOLO v3-tiny achieves speed by sacrificing some accuracy compared to the full YOLO v3 model. It's a smaller and a faster variant of the YOLO v3 model.

It consists of the following:

- 13 Convolutional Layers
- 6 Maxpool Layers
- 2 Route Layers
- 2 YOLO Layers
- 1 Up-sampling Layer (Pranav Adarsh, 2020)



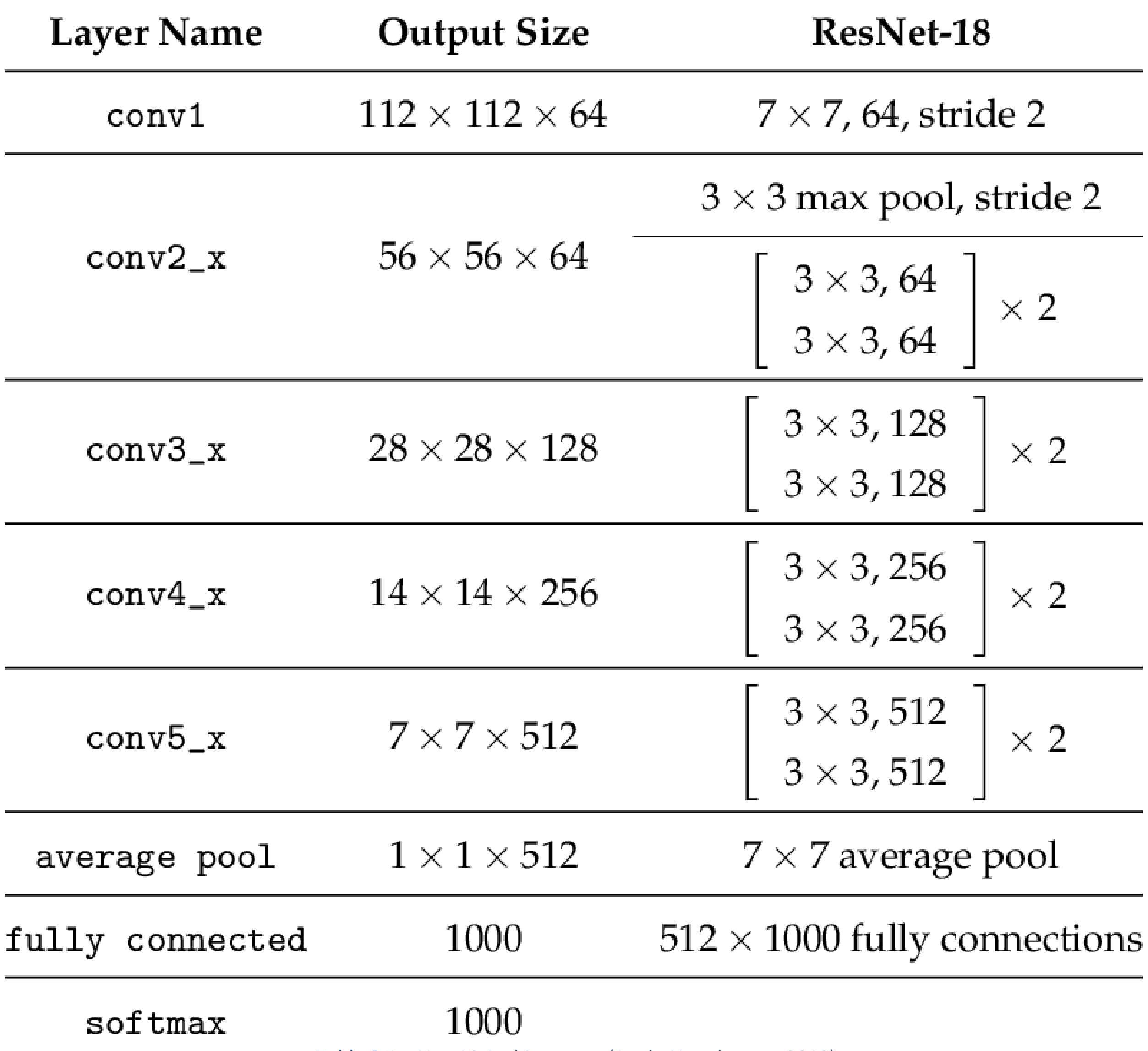
Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

Table 1-Yolo v3-tiny Architecture. *(Yijun He, 2019)*

The table above also describes each layer in depth, including the number of filters in the convolution operation related to that of convolutional layers, the size & stride, the input size to each layer, and the respective output size.

ResNet-18:

The family of ResNet are DNNs designed for deep learning tasks, particularly image classification. They have several types each with different number of layers. E.g. ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152, etc.
In our work, we also use ResNet-18. It consists of 18 layers and is known for its skip connections or residual connections. Skip connections are a technique in neural network architectures that allow the output of one layer to bypass one or more intermediate layers and directly connect to a deeper layer.

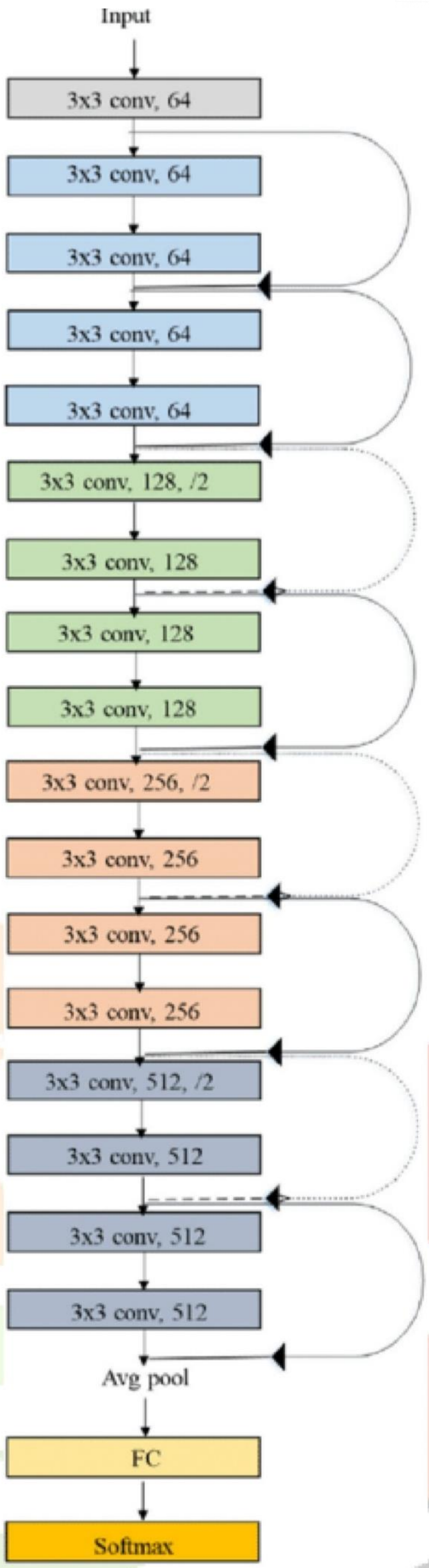


Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

Table 2-ResNet-18 Architecture. *(Paolo Napoletano, 2018)*

It consists of the following:

- 17 Convolutional Layers
- 1 Fully Connected Layer Followed by a SoftMax Activation Function (Resnet18 Model With Sequential Layer For Computing Accuracy On Image Classification Dataset)



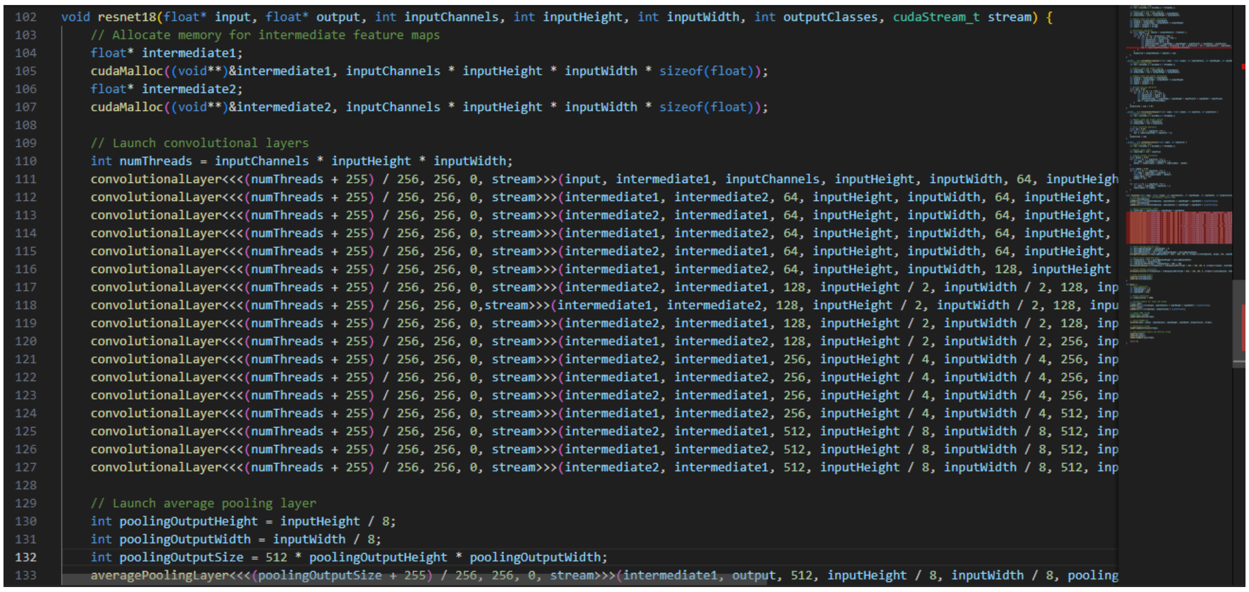
Input
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Avg pool
FC
Softmax

Figure 13-ResNet-18 Architecture.

The table and figure above describe ResNet-18's layers in depth, including a basic visualization of the skip connection technique in Figure 7, the number of filters in the convolution operation related to that of convolutional layers, the size & stride, and the respective output size.

CUDA files (input) of our experiment to the DTM Generator:

Below you can find the .cu files of YOLO v3-tiny and ResNet-18, both having 1 stream.

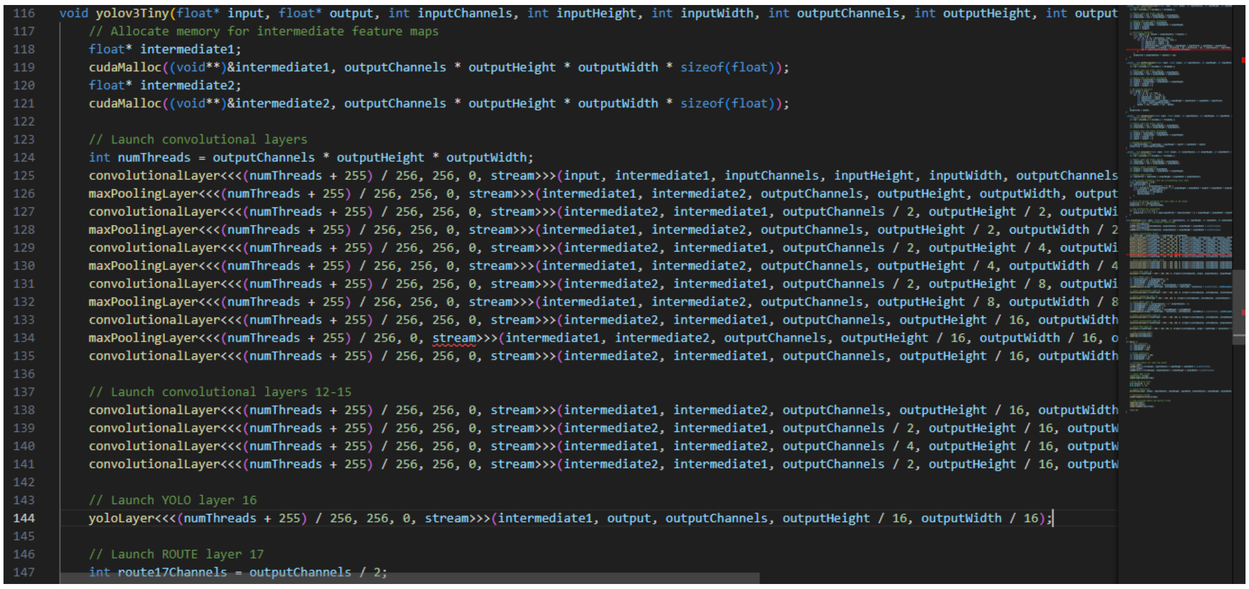


```
void resnet18(float* input, float* output, int inputChannels, int inputHeight, int inputWidth, int outputClasses, cudaStream_t stream) {
    // Allocate memory for intermediate feature maps
    float* intermediate1;
    cudaMalloc((void**)&intermediate1, inputChannels * inputHeight * inputWidth * sizeof(float));
    float* intermediate2;
    cudaMalloc((void**)&intermediate2, inputChannels * inputHeight * inputWidth * sizeof(float));

    // Launch convolutional layers
    int numThreads = inputChannels * inputHeight * inputWidth;
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(input, intermediate1, inputChannels, inputHeight, inputWidth, 64, inputHeigh
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 64, inputHeight, inputWidth, 64, inputHeight,
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 64, inputHeight, inputWidth, 64, inputHeight,
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 64, inputHeight, inputWidth, 64, inputHeight,
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 64, inputHeight, inputWidth, 64, inputHeight,
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 64, inputHeight, inputWidth, 128, inputHeight
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 128, inputHeight / 2, inputWidth / 2, 128, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0,stream>>>(intermediate1, intermediate2, 128, inputHeight / 2, inputWidth / 2, 128, inpu
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 128, inputHeight / 2, inputWidth / 2, 128, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 128, inputHeight / 2, inputWidth / 2, 256, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 256, inputHeight / 4, inputWidth / 4, 256, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 256, inputHeight / 4, inputWidth / 4, 256, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 256, inputHeight / 4, inputWidth / 4, 256, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 256, inputHeight / 4, inputWidth / 4, 512, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 512, inputHeight / 8, inputWidth / 8, 512, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, 512, inputHeight / 8, inputWidth / 8, 512, inp
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, 512, inputHeight / 8, inputWidth / 8, 512, inp

    // Launch average pooling layer
    int poolingOutputHeight = inputHeight / 8;
    int poolingOutputWidth = inputWidth / 8;
    int poolingOutputSize = 512 * poolingOutputHeight * poolingOutputWidth;
    averagePoolingLayer<<<(poolingOutputSize + 255) / 256, 256, 0, stream>>>(intermediate1, output, 512, inputHeight / 8, inputWidth / 8, pooling
```

Figure 14-ResNet-18 CUDA File (1 Stream).



```
void yolov3Tiny(float* input, float* output, int inputChannels, int inputHeight, int inputWidth, int outputChannels, int outputHeight, int output
    // Allocate memory for intermediate feature maps
    float* intermediate1;
    cudaMalloc((void**)&intermediate1, outputChannels * outputHeight * outputWidth * sizeof(float));
    float* intermediate2;
    cudaMalloc((void**)&intermediate2, outputChannels * outputHeight * outputWidth * sizeof(float));

    // Launch convolutional layers
    int numThreads = outputChannels * outputHeight * outputWidth;
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(input, intermediate1, inputChannels, inputHeight, inputWidth, outputChannels
    maxPoolingLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight, outputWidth, output
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels / 2, outputHeight / 2, outputWi
    maxPoolingLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight / 2, outputWidth / 2
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels / 2, outputHeight / 4, outputWi
    maxPoolingLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight / 4, outputWidth / 4
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels / 2, outputHeight / 8, outputWi
    maxPoolingLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight / 8, outputWidth / 8
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels, outputHeight / 16, outputWidth
    maxPoolingLayer<<<(numThreads + 255) / 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight / 16, outputWidth / 16, o
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels, outputHeight / 16, outputWidth

    // Launch convolutional layers 12-15
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels, outputHeight / 16, outputWidth
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels / 2, outputHeight / 16, outputW
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, intermediate2, outputChannels / 4, outputHeight / 16, outputW
    convolutionalLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate2, intermediate1, outputChannels / 2, outputHeight / 16, outputW

    // Launch YOLO layer 16
    yoloLayer<<<(numThreads + 255) / 256, 256, 0, stream>>>(intermediate1, output, outputChannels, outputHeight / 16, outputWidth / 16);

    // Launch ROUTE layer 17
    int route17Channels = outputChannels / 2;
```

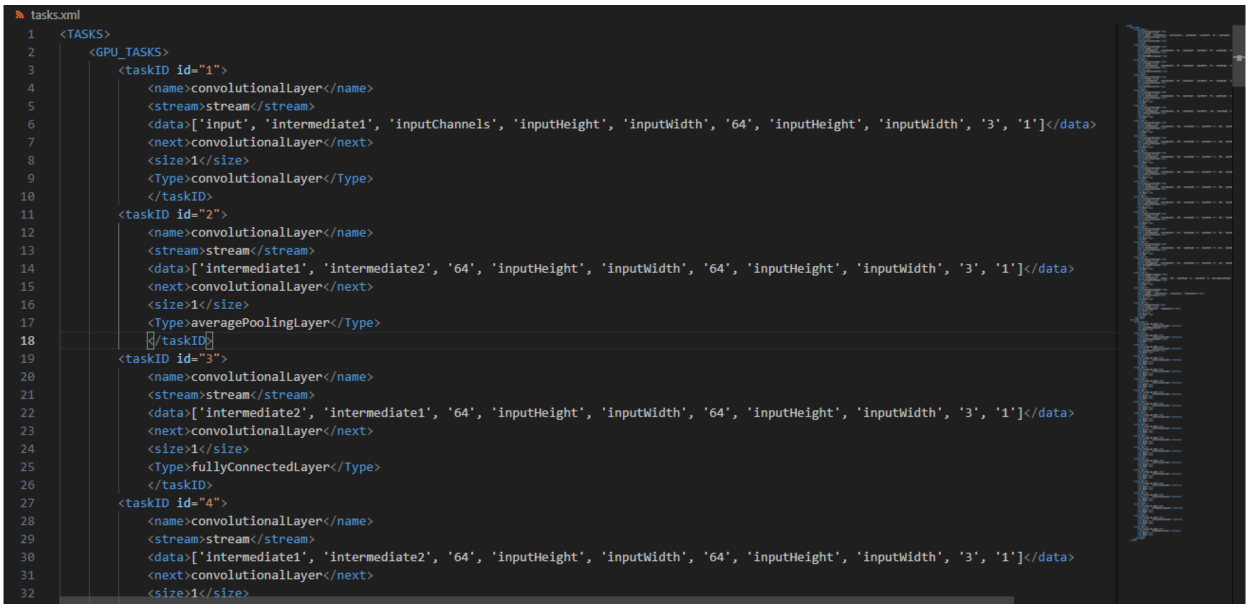
Figure 15-YOLO v3-tiny CUDA File (1 Stream).

In both cases (ResNet-18 & YOLO v3-tiny), we have made approaches with various CUDA files by varying the number of streams: 1, 2, & 4.

4.6 Results of our experiment (output):

We have deployed the CUDA input files to our DTM Generator and we execute the python code in Visual Studio Code. We get the following .xml files as output results of our experiment:

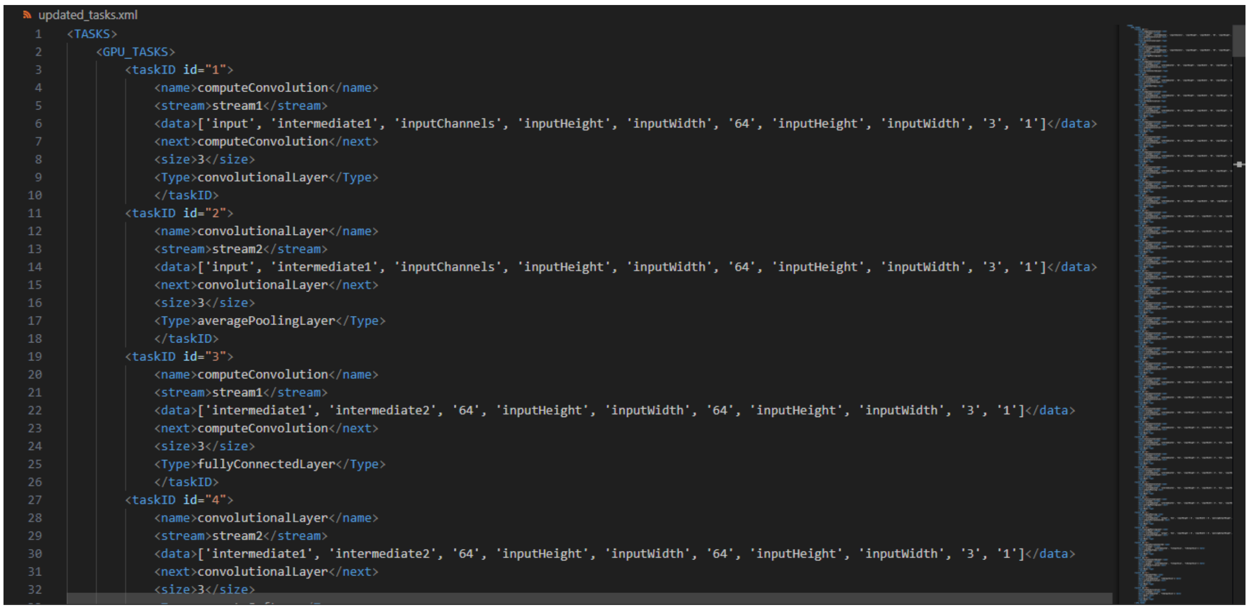
1. ResNet-18:



tasks.xml

```xml
<TASKS>
    <GPU_TASKS>
        <taskID id="1">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['input', 'intermediate1', 'inputChannels', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>1</size>
            <Type>convolutionalLayer</Type>
            </taskID>
        <taskID id="2">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['intermediate1', 'intermediate2', '64', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>1</size>
            <Type>averagePoolingLayer</Type>
            </taskID>
        <taskID id="3">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['intermediate2', 'intermediate1', '64', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>1</size>
            <Type>fullyConnectedLayer</Type>
            </taskID>
        <taskID id="4">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['intermediate1', 'intermediate2', '64', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>1</size>
```

Figure 16-XML File 1 Stream

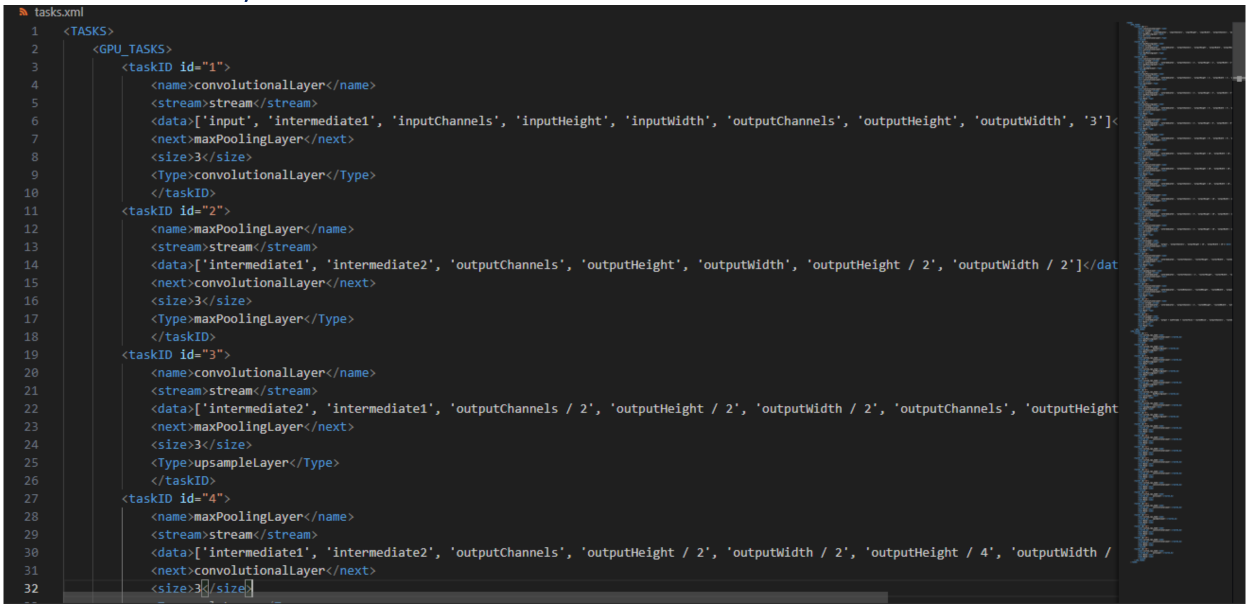


updated_tasks.xml

```xml
<TASKS>
    <GPU_TASKS>
        <taskID id="1">
            <name>computeConvolution</name>
            <stream>stream1</stream>
            <data>['input', 'intermediate1', 'inputChannels', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>computeConvolution</next>
            <size>3</size>
            <Type>convolutionalLayer</Type>
            </taskID>
        <taskID id="2">
            <name>convolutionalLayer</name>
            <stream>stream2</stream>
            <data>['input', 'intermediate1', 'inputChannels', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>3</size>
            <Type>averagePoolingLayer</Type>
            </taskID>
        <taskID id="3">
            <name>computeConvolution</name>
            <stream>stream1</stream>
            <data>['intermediate1', 'intermediate2', '64', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>computeConvolution</next>
            <size>3</size>
            <Type>fullyConnectedLayer</Type>
            </taskID>
        <taskID id="4">
            <name>convolutionalLayer</name>
            <stream>stream2</stream>
            <data>['intermediate1', 'intermediate2', '64', 'inputHeight', 'inputWidth', '64', 'inputHeight', 'inputWidth', '3', '1']</data>
            <next>convolutionalLayer</next>
            <size>3</size>
```

Figure 17-XML File 2 Streams

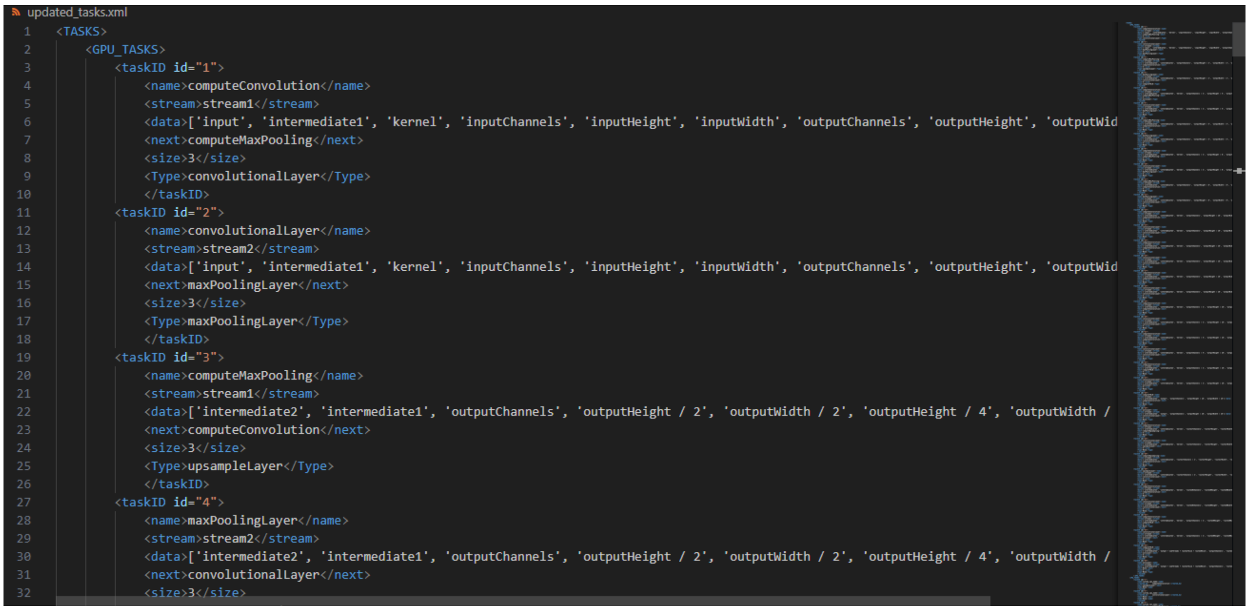
2. YOLO v3-tiny:



tasks.xml

```
<TASKS>
    <GPU_TASKS>
        <taskID id="1">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['input', 'intermediate1', 'inputChannels', 'inputHeight', 'inputWidth', 'outputChannels', 'outputHeight', 'outputWidth', '3']<
            <next>maxPoolingLayer</next>
            <size>3</size>
            <Type>convolutionalLayer</Type>
            </taskID>
        <taskID id="2">
            <name>maxPoolingLayer</name>
            <stream>stream</stream>
            <data>['intermediate1', 'intermediate2', 'outputChannels', 'outputHeight', 'outputWidth', 'outputHeight / 2', 'outputWidth / 2']</dat
            <next>convolutionalLayer</next>
            <size>3</size>
            <Type>maxPoolingLayer</Type>
            </taskID>
        <taskID id="3">
            <name>convolutionalLayer</name>
            <stream>stream</stream>
            <data>['intermediate2', 'intermediate1', 'outputChannels / 2', 'outputHeight / 2', 'outputWidth / 2', 'outputChannels', 'outputHeight
            <next>maxPoolingLayer</next>
            <size>3</size>
            <Type>upsampleLayer</Type>
            </taskID>
        <taskID id="4">
            <name>maxPoolingLayer</name>
            <stream>stream</stream>
            <data>['intermediate1', 'intermediate2', 'outputChannels', 'outputHeight / 2', 'outputWidth / 2', 'outputHeight / 4', 'outputWidth /
            <next>convolutionalLayer</next>
            <size>3</size>
```

Figure 18-XML File 1 Stream



updated_tasks.xml

```
<TASKS>
    <GPU_TASKS>
        <taskID id="1">
            <name>computeConvolution</name>
            <stream>stream1</stream>
            <data>['input', 'intermediate1', 'kernel', 'inputChannels', 'inputHeight', 'inputWidth', 'outputChannels', 'outputHeight', 'outputWid
            <next>computeMaxPooling</next>
            <size>3</size>
            <Type>convolutionalLayer</Type>
            </taskID>
        <taskID id="2">
            <name>convolutionalLayer</name>
            <stream>stream2</stream>
            <data>['input', 'intermediate1', 'kernel', 'inputChannels', 'inputHeight', 'inputWidth', 'outputChannels', 'outputHeight', 'outputWid
            <next>maxPoolingLayer</next>
            <size>3</size>
            <Type>maxPoolingLayer</Type>
            </taskID>
        <taskID id="3">
            <name>computeMaxPooling</name>
            <stream>stream1</stream>
            <data>['intermediate2', 'intermediate1', 'outputChannels', 'outputHeight / 2', 'outputWidth / 2', 'outputHeight / 4', 'outputWidth /
            <next>computeConvolution</next>
            <size>3</size>
            <Type>upsampleLayer</Type>
            </taskID>
        <taskID id="4">
            <name>maxPoolingLayer</name>
            <stream>stream2</stream>
            <data>['intermediate2', 'intermediate1', 'outputChannels', 'outputHeight / 2', 'outputWidth / 2', 'outputHeight / 4', 'outputWidth /
            <next>convolutionalLayer</next>
            <size>3</size>
```

Figure 19-XML File 2 Streams

4.7 Evaluation:

We measured the execution time for generating a Task Model in case of single and double streams. (1 stream, 2 streams) in both DNNs.

The execution time is the time needed for the program to execute and generate the .xml output file. The table below shows the results.

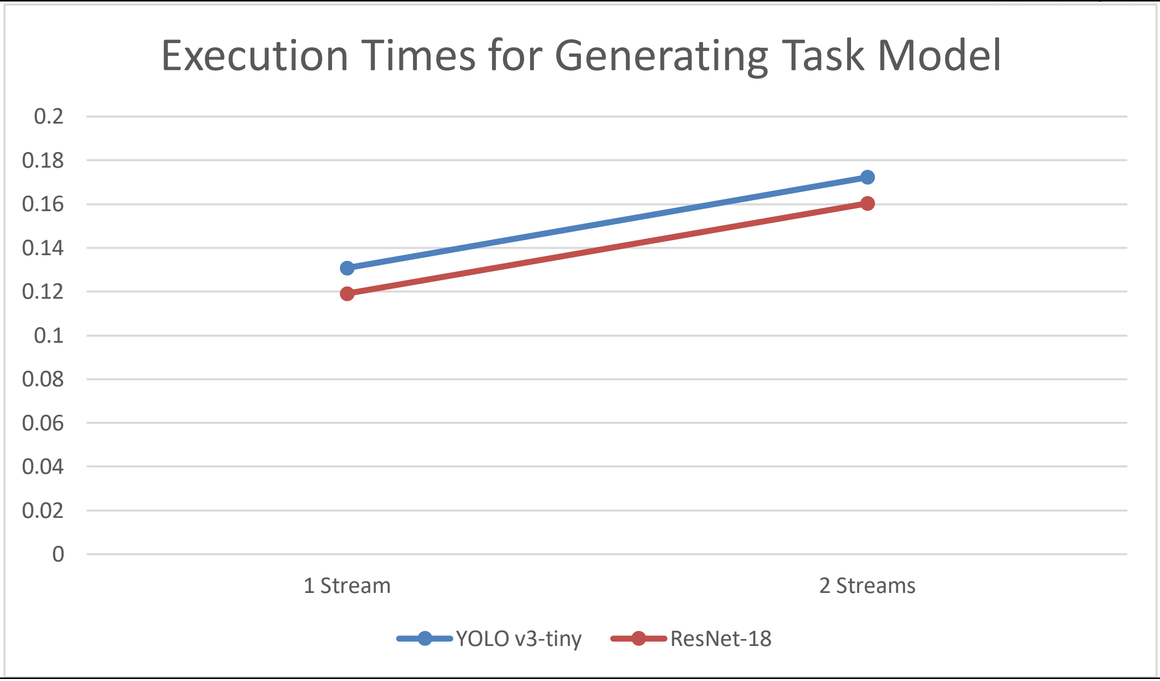
	1 Stream	2 Streams
YOLO v3-tiny	0.1308 s	0.1722 s
ResNet-18	0.1191 s	0.1603 s

Table 3-Computation Time for Task Model Generation using 1 & 2 Streams.

From the graph above, we can notice that the computation time increases with the increase of number of Streams in our CUDA file. While we can make use of the GPU parallel computation and execute several tasks at the same time of our real-time DNN, the computation time for generating the task model increases.

A single stream also makes it very simple to write and edit the CUDA file. However, writing a CUDA file with multiple streams and managing them can add complexity to our CUDA code.

5. Conclusion & Future Work:

In this internship, we introduced DTMG, a DNN Task Model Generator which generates automatically the task model for real-time DNN deployed over Nvidia GPU. From CUDA and specification source files, we are now able to generate task models automatically in xml format. We consider the usage of multiple streams and single stream as well as the CPU-GPU architecture. In DTMG, we take into account all of these factors in order to assess the schedulability of tasks.

Our goal is to decide the schedulablity of the system. Therefore, in future work, we aim to integrate the xml file in *Cheddar*, which is a real-time software tool. It takes xml-format file as input, and it is used for conducting scheduling analysis. Consequently, we will be able to use our generated task model by using *Cheddar*. The DTMG developed in this internship represents a link between the source files containing parameters and *Cheddar*.

6. References

- An Zou, J. L. (2023). Real-time GPU Scheduling of Hard Deadline Parallel Tasks with Fine-Grain Utilization.
- Dehua Zheng, W. Z. (January, 2021). Short-term renewable generation and load forecasting in microgrids. In *Microgrid Protection and Control* (p. 4.4.3.4 Deep neural network).
- Giorgio Buttazzo, G. L. (2005). Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science). Plenum Publishing Co.
- Insup Lee, J. Y. (2007). Handbook of real-time and embedded systems. CRC Press.
- Jamshed, S. (December, 2015). Graphics Processing Unit Technology. In *Using HPC for Computational Fluid Dynamics.*
- Layland, C. L. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM),*, 46-61.
- Morrison, A. (2023). NVIDIA CUDA ARCHITECTURE.
- N. Audsley, a. A. (n.d.). REAL-TIME SYSTEM SCHEDULING. *University of York, UK*.
- Neil Audsley, A. B. (1995). *Realtime system scheduling.* Springer Berlin Heidelberg.
- Paolo Napoletano, F. P. (2018). Anomaly Detection in Nanofibrous Materials by CNN-Based Self-Similarity.
- Pathshala, e.-P. (n.d.). *Embedded System. Module: RTOS: Multiple tasks and Processes. Subject: Computer Science.*
- Pranav Adarsh, M. K. (2020). YOLO v3-Tiny: Object Detection and Recognition using one stage improved model.
- Ranjeet Kaur, A. D. (2022). Brain tumor segmentation using deep learning: taxonomy, survey and challenges. 14.6.1 Basic working of deep neural network.
- Reinhard Wilhelm, J. E. (2008). The worst-case execution-time problem—overview of methods and survey of tools. In *ACM Transactions on Embedded Computing Systems (TECS).*
- Resnet18 Model With Sequential Layer For Computing Accuracy On Image Classification Dataset. (n.d.). *International Journal of Creative Research Thoughts (IJCRT)*, 3.
- Weiguang Pang, X. L. (2023). Efficient CUDA stream management for multi-DNN real-time inference on embedded GPUs. *Journal of Systems Architecture*.
- Yijun He, R. H. (2019). TF-YOLO: An Improved Incremental Network for Real-Time Object Detection.