MotePy: A domain specific language for low-overhead machine learning and data processing.

Jayaraj Poroor
JIFFY.ai
jayaraj.poroor@gmail.com

Abstract—A domain specific language (DSL), named MotePy is presented. The DSL offers a high level syntax with low overheads for ML/data processing in time constrained or memory constrained systems. The DSL-to-C compiler has a novel static memory allocator that tracks object lifetimes and reuses the static memory, which we call the compiler-managed heap.

I. INTRODUCTION

When writing code for machine learning or data processing applications we want the convenience of a high level syntax of a language like Python [1]. Typically this convenience comes with hidden overheads like interpreted execution (e.g., in CPython [2]) or dynamic memory allocation and garbage collection. In many situations these overheads do not pose a concern for the application developer.

However, in certain situations when we have tight time constraints [3] or memory constraints [4], we cannot compromise on the need to have very low overheads. Examples include machine learning pipelines in an OS kernel [5] or in small embedded devices [6].

The proposed DSL, MotePy offers a high level Python-like syntax for specifying modular data processing pipelines and performing vector/matrix operations.

II. SYNTAX

Machine learning and data processing applications typically have a pipeline-like execution structure. The main module of a MotePy application is a pipeline specification of the form given in the example below (Figure 1). The pipeline specification is a Python-like list of operations. In the example, the first operation is acquire function defined in the module datasource and the second operation is predict function defined in model module. MotePy’s runtime executes the pipeline in a loop, starting from the first operation, in sequence till the last operation. Once the last operation is complete the pipeline is re-executed from the beginning, in a loop fashion.

Figure 2 shows the datasource module and Figure 3 shows the model module used in the sample pipeline in Figure 1.

* This work was supported by a grant from Microsoft Research, India.

```python
pipeline.py
[
 datasource.acquire,
 model.predict
]
```

```python
datasource.py
```data: float[60]
def init() -> void:
    pass
@flow
def acquire() -> void:
    for i in range(0, 60):
        data[i] = float(rand())/RAND_MAX
    printf("Next\n")
next(data)
```

Fig. 1. A sample pipeline specification

Fig. 2. The datasource module used in the pipeline specification in Figure 1

Though the syntax is close to Python, the language differs from Python in its implementation in significant ways:

- MotePy is statically typed. Its type annotation syntax is based on Python’s optional type annotation syntax, but uses MotePy’s own notation similar to that of Java for array typing. The reason for own notation for array typing is that type annotations are not common in Python and Java’s array notation is familiar to most programmers.
- MotePy supports only static memory allocation. Only primitive types are allocated on the stack. Rest are allocated in the static data area. However, unlike other languages such as C or C++, the MotePy compiler tracks lifetimes of array objects and reuses memory. We call this the compiler-managed heap.

Every MotePy module has an init function and a special flow function. The flow function is the operation invoked as part of the pipeline execution and is decorated with @flow syntax. Apart from these two functions a
model.py

N : const int32_t = 60
M : const int32_t = 30

layer0 : float[N][M]
layer1 : float[M]

def random_weights() -> void:
    for j in range(0, M):
        for i in range(0, N):
            layer0[i][j] = float(rand()) / RAND_MAX
            layer1[j] = float(rand()) / RAND_MAX

def init() -> void:
    random_weights()

@flow
def predict(features: float[N]) -> void:
    res: float
tmp : float[30]
printf("Prediction\n")
tmp = features * layer0
res = tmp * layer1
#res = exp(res)
printf("Result␣%f\n", res)

Fig. 3. The model module used in the pipeline specification in Figure 1

module may define any number of functions to perform its function.

In Figure 2 the data array is randomly initialized and then passed onto the next stage in the pipeline by invoking a special next function. The next function enables us to specify at what point in the execution the next stage in pipeline must be executed. If a flow function returns without invoking the next function the pipeline is re-executed from the beginning. For instance, if the data acquisition is not complete, then the acquisition stage may return without calling next. The MotePy runtime will automatically re-execute the pipeline.

Figure 3 shows a neural-network-like processing stage. The network has two layers of weights that are randomly initialized in the example. In an actual system the weights will be loaded from a file. The code shows high level syntax for vector/matrix operations, which are translated by the MotePy compiler into low-level loop operations.

The code also shows invoking the C function printf. MotePy allows us to invoke C/C++ functions from MotePy code without any need for a special foreign function calling interface. The MotePy array/matrix layout is same as that of C/C++ enabling us to pass arrays/matrices back and forth between C/C++ code without any transformation overheads.

The MotePy compiler translates the high level DSL code to low level C/C++ code, which is then compiled using a platform C/C++ compiler such as GCC to get the executable.

III. COMPILER-MANAGED HEAP

Figure 4 shows a two-stage MotePy pipeline. Stage 1 initializes vec1, does some computations on it and then invokes next for Stage 2 processing. Stage 2 initializes vec2, does some computations on it, then initializes vec3 and does some computations on it.

Compiling and running the program gives the following output:

Address of vec1: 94473766309984
Address of vec2: 94473766309984
Address of vec3: 94473766309984

The output shows that the addresses of vec1, vec2, and vec3 are same\(^1\). The reason is that the compiler allocates static memory for the vectors and reuses the memory based on the variable lifetimes. It may be noted that owing to Address Space Layout Randomization\(^7\) the printed addresses will be different during each program execution.

In the case of this example, the life time of vec1 is in the program region from line numbers 8-11 in stage1.py. Life time of vec2 is from line numbers 8 - 11 in stage2.py and that of vec3 is from line numbers 12 - 15 in stage2.py. Since the lifetimes are non-overlapping the compiler intelligently reuses the static memory area for the three vectors.

Figure 5 shows a modified version of the Stage 1 of the pipeline. Here the vec1 is initialized in the init function. The process function computes a new value of vec1 based on the current value. Compiling and running the pipeline with this modified Stage 1 gives the following result:

Address of vec1: 94035385520224
Address of vec2: 94035385520464
Address of vec3: 94035385520464

The output shows that the compiler allocates a separate non-overlapping memory area for vec1 and uses the same memory area for vec2 and vec3. The compiler’s allocator now knows that vec1’s lifetime overlaps with that of vec2 and vec3 and hence allocates separate memory for vec1.

MotePy’s static allocator allocates along the 2-dimensional space with address as one axis and variable lifetime as the other axis. In comparison, the static allocators of languages like C/C++ simply allocates along the address line.

\(^1\)The generated executable has an optional parameter for specifying the number of pipeline iterations. In this case a value of ‘1’ was passed. If none is passed, the pipeline is executed in an infinite loop.
pipeline.py

```python
[
    stage1.process,
    stage2.process
]
```

stage1.py

```python
1. vec1: float[60]
2. 3. def init() -> void:
4.    vec1 = 0
5.    pass
6. 7. def process() -> void:
8.     vec1 = vec1 + vec1 #some processing
9.     printf("Address of vec1: %ld\n", 
        10.     uint64_t(vec1))
11. next()
```

stage2.py

```python
1. vec2: float[60]
2. vec3: float[60]
3. 4. def init() -> void:
5.     pass
6. 7. def process() -> void:
8.     vec2 = vec2 + vec2
9.     printf("Address of vec2: %ld\n", 
        10.     uint64_t(vec2))
11. vec3 = 0
12. 13. vec3 = vec3 + vec3
14.     printf("Address of vec3: %ld\n", 
        15.     uint64_t(vec3))
16. next()
```

Fig. 4. A two-stage MotePy pipeline

Fig. 5. A modified version of Stage 1

Surya Chaitanya who worked with me as student interns during the initial development of this work.

References

[1] Rossum, Guido. "Python reference manual." (1995).
[2] Roghult, Alexander. "Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy." (2016).
[3] Liu, Fan, Ajit Narayanan, and Quan Bai. "Real-time systems." (2000).
[4] Lee, Edward A. "Cyber-physical systems-are computing foundations adequate." Position paper for NSF workshop on cyber-physical systems: research motivation, techniques and roadmap. Vol. 2. Citeseer, 2006.
[5] Negi, Atul, and P. Kishore Kumar. "Applying machine learning techniques to improve linux process scheduling." TENCON 2005-2005 IEEE Region 10 Conference. IEEE, 2005.
[6] Dennis, Don Kurian, et al. "EdgeML Machine LEARNING for Resource-Constrained Edge Devices." URL https://github.com/Microsoft/EdgeML. Retrieved January (2020).
[7] Jang, Yeongjin, Sangho Lee, and Tae soo Kim. "Breaking kernel address space layout randomization with intel tsx." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016.

IV. ACKNOWLEDGEMENTS

I would like to thank Microsoft Research, India for the grant support. I am especially thankful to Dr. Srimram Rajamani, Satish Sangameswaran, and Dr. Harsha Vardhan Simhadri for the support during the course of this work. I thank Nandagopal, Sandesh Ghanta, and...