PyOMP: Parallel programming for CPUs and GPUs with OpenMP and Python

Giorgis Georgakoudis*
georgakoudis1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Todd A. Anderson*
todd.a.anderson@intel.com
Intel Research Labs
Hillsborough, Oregon, USA

Stuart Archibald
sarchibald@anaconda.com
The Numba project
(sponsored by Anaconda Inc.)
United Kingdom

Bronis de Supinski
desupinski1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Timothy G. Mattson
tim@timmattson.com
Human Learning Group
Ocean Park, Washington, USA

Abstract

Python is the most popular programming language. OpenMP is the most popular parallel programming API. Projecting OpenMP into Python will help expand the HPC community. We call our Python-based OpenMP system *PyOMP*.

In this short paper we describe PyOMP and its use for parallel programming for CPUs and GPUs. We describe its implementation through the well known Numba just-in-time (JIT) compiler and how to install PyOMP on your own systems. We provide some performance results suggesting performance on par with that from C and OpenMP, but our focus here is not detailed benchmarking. We leave that to other papers. Our goal here is to show how to use PyOMP so we can grow the PyOMP community.

Keywords

Python, OpenMP, Parallel Programming, PyOMP

1 Introduction

Python is the world's most popular programming language [2, 3]. OpenMP is the world's most popular parallel programming model [7]. To make Python a more effective language for HPC, we need OpenMP inside Python.

We recently released PyOMP [8], a system that maps OpenMP into Python using the Numba Just-In-Time (JIT) compiler. Originally, PyOMP only supported parallel programming for a CPU. In this paper, we introduce our extension of PyOMP to support parallel programming for a GPU.

Given hardware trends with tightly integrated CPUs and GPUs, programming CPUs and GPUs through a single API is important. This will let code move between the CPU and the GPU as we map each portion of our applications onto the most suitable hardware. The fact PyOMP supports both CPU and GPU programming could turn out to be an extremely important capability.

This is a short paper, so we need to be concise in our presentation. We will start by introducing PyOMP in enough detail so people can start using it. We will take a simple program and show how it can be parallelized using the key design patterns for parallel programming. Our goal with this paper is to build a user community around PyOMP, so we close the paper with a detailed description of how to install PyOMP on your own systems.

*Both authors contributed equally to this research.

2 PyOMP: A Pythonic OpenMP User Interface

Our goal is a Python interface to OpenMP that is *pythonic*. OpenMP is a directive driven system that supports a style of parallel programming where parallelism is added incrementally to a working program. In C/C++ this is done with pragmas. What is the Pythonic analog to pragmas for communicating with the runtime/compilation system?

We found [4] that the `with` statement provides the behavior we needed. The Python `with` statement invokes a context manager to open a resource, carry out operations using that resource, and then close that resource. This can apply to a team of threads on a CPU or to a kernel offloaded onto a GPU. For example, the following code will create a team of threads (the resource) to carry out a vector addition in PyOMP.

```
with openmp("parallel for"):
    for i in range(N):
      C[i] = A[i]+B[i]
```

At the end of the context statement, the team of threads are joined so only a single threads proceeds. A vector addition on a GPU would be accomplished with very similar code using PyOMP.

```
with openmp("target teams loop"):
    for i in range(N):
      C[i] = A[i]+B[i]
```

The `target` construct moves the work to the GPU while the `teams loop` construct maps that work (the loop iterations) onto the index space defined by the loop bounds. With `loop` you leave all details of how the work distribution is managed to the OpenMP system. Lower level constructs are available (as we describe later) when more detailed control is needed. In our view, this design is pythonic, easy to use, and lets one seamlessly move between CPU parallelism and GPU parallelism within the same program.

The key to implementing PyOMP is the Numba JIT compiler. You decorate the function containing pyomp context statements with the `@njit` decorator. Numba will JIT compile the code and cache it for later use. This means you only pay the cost for JIT compilation once. The resulting code targets LLVM. By using a

version of LLVM that includes an OpenMP runtime library, we are able exploit the full functionality of OpenMP in our generated LLVM code. This has the further advantage of bypassing Python's Global Interpreter Lock (GIL) to achieve parallel performance from multithreaded execution.

In the following sections we will explore in more detail how to use PyOMP to write multithreaded code and to write code that executes on a GPU. We will do this by looking at the key design patterns used in parallel computing. By design, if you know programming with OpenMP for C, C++ or Fortran you know programming in PyOMP.

2.1 PyOMP: CPU programming

Most programming of the CPU uses one of three patters: the Single Program Multiple Data (SPMD) pattern, (2) the loop level parallelism pattern, or the Divide and conquer pattern (with tasks). Following closely the discussion from our SciPy paper [4], we describe PyOMP for CPU programming by considering a simple program (numerical integration to approximate π) implemented in each of these patterns.

2.1.1 The SPMD Pattern. The Single Program Multiple Data Pattern (SPMD) is probably the most commonly used design pattern in all of parallel computing. We show a program using that pattern in figure 1. In an SPMD pattern, you create a team of threads and then, using the rank of a thread (a number ranging from zero to the number of threads minus one) and the number of threads, explicitly control how work is divided among the threads. Threads are created with the *parallel* construct expressed in PyOMP using the with context statement. The identifier openmp indicates this is an element of PyOMP and parallel indicates that the compiler should fork a *team* of threads. These threads come into "existence" at that point in the program and they each redundantly execute the work in the region associated with the with statement.

OpenMP is a shared memory API. The threads "belong" to a single process and they all share the memory associated with the process. Variables visible outside a parallel construct are by default shared inside the construct. Variables created inside a construct are by default *private* to the construct (i.e., there is a copy of the variable for each thread in the team). It is good form in OpenMP programming to make the status of variables explicit in an OpenMP construct which we do with the shared and private clauses in figure 1.

In an SPMD program, you need an ID (or `thread_num`) of each thread and number of threads (`num_threads`) in the team. We do this with OpenMP runtime functions. All threads in a single team see the same value for the number of threads (`nThrds`) so this is a shared variable. In multithreaded programming, it is a data race if multiple threads write to the same variable; even if the value being written is the same for each thread. So we must assure that only one thread sets the value for the number of threads. This is done with a single construct. The other threads wait until the thread executing the single construct is done before any threads execute the following code.

Distribution of work between threads is handled through the for-loop. Each thread starts with a loop iteration (i) equal to its rank, which is incremented by the number of threads. The result is

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_thread_num
from numba.openmp import omp_get_num_threads
MaxTHREADS = 32

@njit
def piFunc(NumSteps):
  step = 1.0/NumSteps
  partialSums = np.zeros(MaxTHREADS)
  with openmp('''parallel shared(partialSums,nThrds)
              private(threadID,i,x,localSum)'''):
     threadID = omp_get_thread_num()
     with openmp("single"):
        nThrds = omp_get_num_threads()

     localSum = 0.0
     for i in range(threadID,NumSteps,nThrds):
         x = (i+0.5)*step
         localSum = localSum + 4.0/(1.0 + x*x)
     partialSums[threadID] = localSum

  pi = step*np.sum(partialSums)
  return pi
pi = piFunc(100000000)
```

Figure 1: A program using the SPMD pattern to numerically approximate a definite integral that should equal π.

loop iterations dealt out as if from a deck of cards. This commonly used technique is called a "cyclic distribution of loop iterations". This loop sums values of the integrand which we accumulate into a location local to each thread. Since we need to later combine these local sums to get the final answer, we store the local sum into a shared array (partialSums).

When the parallel region ends the team of threads join together and the original thread continues. We show runtimes for this SPMD program in table 1. We do not include the time for the JIT compilation. This is usually justified since in a full application, a function is JIT'ed once and then called many times. As we can see in the table, the runtimes for PyOMP and C/OpenMP are comparable.

2.1.2 Loop Level Parallelism. The Loop Level Parallelism pattern is the best known pattern for OpenMP. This is shown in figure 2. Parallelism is introduced through a single with statement to express the parallel for construct. This construct creates a team of threads and then distributes the iterations of the loop among the threads. To accumulate the summation across loop iterations, we include the reduction clause. This clause defines reduction with the + operator over the variable sum. A copy of this variable is created for each thread in the team. It is initialized to the identity for the operator (which in this case is zero). At the end of the loop, all the threads wait for the other threads (a synchronization operation called a *barrier*). Before exiting the barrier, the local copies of sum are combined into a single value, that value is combined with the value of sum from before the parallel loop construct, and the threads join

N	SPMD	loop	task	C SPMD	C loop	C Task
1	0.450	0.447	0.453	0.448	0.444	0.445
2	0.255	0.252	0.245	0.242	0.245	0.222
4	0.164	0.160	0.146	0.149	0.149	0.131
8	0.0890	0.0890	0.0898	0.0826	0.0827	0.0720
16	0.0503	0.0520	0.0517	0.0451	0.0451	0.0431

Table 1: Programs to approximate a definite integral whose value equals π using the SPMD, loop level, and divide-and-conquer/task pattern. Runtimes in seconds for PyOMP and analogous C programs. Note: JIT compilation for the PyOMP runtimes is *not* included. All programs were run on an Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz. C programs used the Intel® icc compiler version 19.1.3.304 as `icc -qnextgen -O3 -fiopenmp`

.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
  step = 1.0/NumSteps
  sum = 0.0
  with openmp("parallel for private(x) reduction(+:sum)"):
      for i in range(NumSteps):
          x = (i+0.5)*step
          sum += 4.0/(1.0 + x*x)

  pi = step*sum
  return pi
pi = piFunc(100000000)
```

Figure 2: A program using a parallel loop to numerically approximate a definite integral that should equal π.

so only the single, original thread continues. The results for this pattern are shown as the second column in table 1. Once again, the performance is similar to that achieved with the C version of the program.

2.1.3 Tasks and Divide and Conquer. Our final pattern is complex and requires more code than can fit in this short paper (all code used in this paper, as well as PyOMP itself, is available in GitHub [1]). It uses the Divide and Conquer pattern based on building a directed Graph of tasks. A wide range of problems including optimization problems, spectral methods, and cache oblivious algorithms use the divide and conquer pattern.

The pattern consists of three phases: split, compute, and merge. The *split phase* recursively divides a problem into smaller subproblems. After enough splits, the subproblems are small enough to directly compute in the *compute phase*. The final phase merges subproblems together to produce the final answer.

An outline of how this works in PyOMP is shown in figure 3. If the work is small enough, the function just does the computation and returns the result (the compute phase). If not, then two tasks are created (the split phase); one for the first half of the work and the other for the second half. Once the tasks are created, that function pauses and waits until the tasks have finished (the merge phase).

```
def doit(work):
  if(small(work)):
      x = do_the_comp(work)
  with openmp("task shared(x1)"):
      x1 = work(first_half(work))
  with openmp("task shared(x2)"):
      x2 = work(second_half(work))
  with openmp("taskwait"):
      x = x1 + x2
  return x
```

Figure 3: The pattern of tasks and taskwait in the recursive function in a divide and conquer pattern.

The tasking code based on this pattern resulted in the runtimes summarized in table 1. Even though the code is more complex than for the other two patterns, the runtimes for this problem are comparable to the other patterns for both Python and C.

2.2 PyOMP: GPU programming

GPU programming with OpenMP [5] was added in OpenMP 4.0 released in 2013. To program a GPU, you define an index space (or a grid). An instance of a function (a *kernel*) executes for each point in that *index space*. All of the instances (*work-items* or *threads*) for a given kernel invocation are organized into groups (*teams*) which taken together define a *league of teams*. The *team* is the unit of scheduling. For each team in a league, all the threads in the team are active on a *compute unit* (or a *streaming multiprocessor*) of a GPU at one time. The teams run on the compute units concurrently with any extra teams (should there be more teams in a league than compute units) enqueued and waiting to execute.

In Figure 4, we summarize the most commonly used constructs and clauses for GPU programming with OpenMP. The fundamental offload construct is `target`. It is used to offload data between the CPU and the GPU by creating (and updating) a *data region* on the GPU. The `target` construct is also used to launch a kernel on the GPU. The `teams` construct creates a league of teams while the `distribute` construct establishes the index space as the iterations of a set of nested loops. These iterations are *distributed* among the league of teams. The kernel itself is the loop-body of the nested loops. Alternatively, as we showed earlier, detailed control of how kernel istances map onto the GPU can be left to the compiler by using a simple `target teams loop` construct.

For the most part, the semantics of constructs used for GPU programming for PyOMP matches those for C/C++/Fortran [5]. There are some details, however, that are different due to the ways Python handles data compared to more traditional HPC languages. Given the scope of this paper, we can't discuss them here. These details, and more, are discussed in depth in the PyOMP repository [1].

As for performance, detailed benchmarking for GPU programming using PyOMP will appear in a paper that is currently under preparation. We considered a subset of programs from the HeCBench [6] program suite. Generally, we found comparable performance with programs implemented with C and OpenMP running on NVIDIA A100 GPUs.

Figure 4: The most commonly used PyOMP directives when programming a GPU with PyOMP. PyOMP constructs as context statements: the first for data movement and the second for computation.

```
with openmp("target data map(to: A,B) map(tofrom: C)"):
  with openmp("target teams distribute parallel for thread_limit(256) "):
    for i in range(N):
        C[i] += A[i] + B[i]
```

Explanation of PyOMP constructs and clauses.

Construct/Clause	Description
target data	Create a data region on the device.
map(to:A,B)	Map arrays A and B to the device. Do not copy back to the host at the end of the data region
map(tofrom:C)	Map array C to the device and copy back to the host from the device at the end of the data region.
target teams	Offload work to the device and launch an initial thread for each team in the league of teams.
distribute	Distribute the associated loop iterations among the league of teams.
parallel for	create a team of threads to execute loop iterations in parallel for each of the teams.
thread_limit(256)	Default number of threads (256) per team (fewer is OK if you run out of work).

3 Installing PyOMP on your own system

In this section, we describe the options for installing PyOMP on one's own system. The easiest way to install PyOMP is using the Conda package manager. We provide PyOMP packages for four different architectures: `linux-ppc64le`, `linux-64`, `osx-arm64`, and `linux-arm64`. Installation is a single command:

```
conda install -c python-for-hpc -c conda-forge pyomp
```

It is also possible to try PyOMP, without a local installation, in a Jupyter notebook using the free cloud-hosted Binder service on a multi-core CPU. We provide a link to launch PyOMP through Binder: PyOMP on Binder.

We also provide containers for the `amd64` and `arm64` architectures as another way of installation-free usage. There are two ways of using PyOMP from the container: (1) through Jupyter exporting a web interface, and (2) through terminal access to the container environment. We show commands to pull and run the container assuming Docker (Podman is also an option).

```
docker pull ghcr.io/python-for-hpc/pyomp:latest
# Jupyter
docker run -it -p 8888:8888 pyomp:latest
# Terminal
docker run -it pyomp:latest /bin/bash
```

4 Conclusion

PyOMP maps OpenMP into Python to support HPC applications. It is "Pythonic" and is a natural way for people to approach parallel programming from within Python.

PyOMP works for CPU and GPU programming. Our benchmarks show that once execution passes from the Python interpreter to machine code (via Numba and LLVM), performance matches that from lower-level programming languages that use the same LLVM/runtime infrastructure. Hence, performance largely matches that from coding with C and OpenMP.

At the same time, we acknowledge that more detailed benchmarking is needed. Benchmarking done right is complex. This is work in progress which will hopefully appear soon in a later paper. Our goal in this paper was not a performance study. Our goal was to make people aware of PyOMP and provide the information needed to get started with PyOMP so we can build a PyOMP community.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy, partially funded by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-870683). This work is also supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

References

[1] [n. d.]. github repository for PyOMP code. https://github.com/Python-for-HPC/PyOMP. [Accessed 25-03-2024].
[2] [n. d.]. PopularitY of Programing Languages. https://pypl.github.io/PYPL.html. [Accessed 25-03-2024].
[3] [n. d.]. TIOBE index. https://www.tiobe.com/tiobe-index/. [Accessed 25-03-2024].
[4] Todd A. Anderson and Timothy G. Mattson. 2021. Multithreaded parallel Python through OpenMP support in Numba. http://conference.scipy.org/proceedings/scipy2021/tim_mattson.html. In *SciPy*.
[5] Tom Deakin and Timothy G. Mattson. 2023. *Programming your GPU with OpenMP*. The MIT Press.
[6] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. https://doi.org/10.1109/ISPASS57527.2023.00041
[7] Tal Kadosh, Niranjan Hasabnis, Timothy G. Mattson, Yuval Pinter, and Gal Oren. 2023. Quantifying OpenMP: Statistical Insights into Usage and Adoption. In *IEEE High Performance Extreme Computing (HPEC)*.
[8] Timothy G. Mattson, Todd A. Anderson, and Giorgis Georgakoudis. 2021. PyOMP: Multithreaded Parallel Programming in Python. *Computing in Science & Engineering* 23, 6 (2021), 77–80. https://doi.org/10.1109/MCSE.2021.3128806