# Cython のスレッド並列処理

この記事は、インテル® デベロッパー・ゾーンに公開されている「Thread Parallelism in Cython*」の日本語参考訳です。

---

## はじめに

Cython は、Python* のスーパーセットで、変数およびクラス属性で C の関数と型をサポートします。Cython は、Python* プログラムの実行を高速化する外部 C ライブラリーのラップに使用します。Cython は、`import` 文によりメイン Python* プログラムで使用される C 拡張モジュールを生成します。

Cython の特長の 1 つは、ネイティブ並列処理 (英語) をサポートすることです (`cython.parallel` モジュールを参照)。`cython.parallel.prange` 関数は並列ループに使用できるため、Python* でスレッド並列処理を使用して、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーを活用することができます。

## インテル® Distribution for Python* 2017 の Cython

インテル® Distribution for Python* 2017 は、NumPy*、SciPy*、Jupyter、matplotlib、Cython、その他のコア Python* パッケージを高速化する、Python* インタープリターのバイナリー・ディストリビューションです。パッケージは、インテル® マス・カーネル・ライブラリー (インテル® MKL)、インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL)、pyDAAL、インテル® MPI ライブラリー、インテル® スレッディング・ビルディング・ブロック (インテル® TBB) を統合します。これらのパッケージに関する詳細は、リリースノート (英語) を参照してください。

インテル® Distribution for Python* 2017 は、こちらからダウンロードできます。Windows® 7 以降、Linux*、OS X* の各オペレーティング・システム向けに、2 つ (Python* 2.7.x および Python* 3.5.x) のパッケージが用意されています。パッケージは、スタンドアロンで、またはインテル® Parallel Studio XE 2017 とともにインストールできます。

インテル® Distribution for Python* は、Python* 2 および Python* 3 をサポートします。インテル® Distribution for Python* には、2 つ (Python* 2.7 および Python* 3.5) のパッケージがあります。この記事では、インテル® Distribution for Python* 2.7 for Linux* (`l_python27_pu_2017.0.035.tgz`) を、インテル® Xeon Phi™ プロセッサー 7250 (68 コア、1.40GHz、コアあたり 4 つのハードウェア・スレッド (合計 272 ハードウェア・スレッド)) にインストールします。以下のように、パッケージのコンテンツを展開した後、インストール・スクリプトを実行してインストールします。

```
$ tar -xvzf l_python27_pu_2017.0.035.tgz
$ cd l_python27_pu_2017.0.035
$ ./install.sh
```

インストールが完了したら、ルート環境をアクティブ化します (リリースノート (英語) を参照)。

```
$ source /opt/intel/intelpython27/bin/activate root
```

## Cython のスレッド並列処理

Python* には、複数のネイティブスレッドが *bycodes* を同時に実行することを防ぐ mutex があります。このため、Python* のスレッドは並列に実行できません。このセクションでは、Cython のスレッド並列処理について説明します。この機能を拡張モジュールとして Python* コードにインポートすると、Python* コードでコアとハードウェア・スレッドをすべて利用することができます。

拡張モジュールを生成するには、Cython コード (拡張子 `.pyx`) を記述します。次に、`.pyx` ファイルを Cython コンパイラーでコンパイルして、C コード (ファイル拡張子 `.c`) に変換します。`.c` ファイルを C/C++ コンパイラーでコンパイルおよびリンクして、共有ライブラリー (`.so` ファイル) を生成します。共有ライブラリーは、モジュールとして Python* にインポートできます。

次の `multithreads.pyx` ファイルで、`serial_loop` 関数は A 配列と B 配列の各エントリーの `log(a)*log(b)` を計算して、C 配列に結果を格納します。log 関数は C 数学ライブラリーからインポートされます。NumPy* モジュール (ハイパフォーマンス科学計算/データ統計パッケージ) は、A 配列と B 配列の操作をベクトル化します。

同様に、`parallel_loop` 関数は OpenMP* スレッドを使用して同じ計算を行い、ループ本体の計算を実行します。`range` を使用する代わりに、`prange` (並列範囲) を使用して複数のスレッドを並列に実行できるようにします。`prange` は `cython.parallel` モジュールの関数で、並列ループに使用できます。この関数が呼び出されると、OpenMP* はスレッドプールを開始し、ワークをスレッド間で分散します。`prange` 関数は、`nogil` コンテキストでループをプットしてグローバル・インタープリター・ロック (GIL) をリリースした場合にのみ使用できることに注意してください (GIL グローバル変数は複数のスレッドを同時に実行しないようにします)。`wraparound(False)` では、Cython は負のインデックスをチェックしません。`boundscheck(False)` では、Cython は配列の境界チェックを行いません。

```
$ cat multithreads.pyx
cimport cython
import numpy as np
cimport openmp
from libc.math cimport log
from cython.parallel cimport prange
from cython.parallel cimport parallel

THOUSAND = 1024
FACTOR = 100
NUM_TOTAL_ELEMENTS = FACTOR * THOUSAND * THOUSAND
X1 = -1 + 2*np.random.rand(NUM_TOTAL_ELEMENTS)
X2 = -1 + 2*np.random.rand(NUM_TOTAL_ELEMENTS)
Y = np.zeros(X1.shape)

def test_serial():
    serial_loop(X1,X2,Y)

def serial_loop(double[:] A, double[:] B, double[:] C):
    cdef int N = A.shape[0]
    cdef int i

    for i in range(N):
        C[i] = log(A[i]) * log(B[i])

def test_parallel():
    parallel_loop(X1,X2,Y)
```

```
@cython.boundscheck(False)
@cython.wraparound(False)
def parallel_loop(double[:] A, double[:] B, double[:] C):
    cdef int N = A.shape[0]
    cdef int i

    with nogil:
        for i in prange(N, schedule=&apos;static&apos;):
            C[i] = log(A[i]) * log(B[i])
```

Cython コードを完了した後、Cython コンパイラーは Cython コードを C コード拡張子ファイルに変換します。この処理は、disutils `setup.py` ファイルにより行われます (disutils は Python* モジュールを分散するために使用されます)。OpenMP* サポートを使用するには、次のように、`setup.py` ファイルのコンパイラー引数とリンク引数で `-fopenmp` オプションを指定して、OpenMP* を有効にするようにコンパイラーに伝える必要があります。`setup.py` ファイルは、拡張モジュールを生成する `setuptools` ビルドプロセスを起動します。デフォルトでは、この `setup.py` は GNU* GCC を使用して Python* 拡張の C コードをコンパイルします。ここでは、`-O0` コンパイラー・オプション (すべての最適化を無効にする) を追加して、ベースラインのパフォーマンスを測定します。

```
$ cat setup.py

from distutils.core import setup
from Cython.Build import cythonize
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
  name = "multithreads",
  cmdclass = {"build_ext": build_ext},
  ext_modules =
  [
    Extension("multithreads",
              ["multithreads.pyx"],
              extra_compile_args = ["-O0", "-fopenmp"],
              extra_link_args=[&apos;-fopenmp&apos;]
              )
  ]
)
```

次のコマンドを使用して、C/C++ 拡張コードをビルドします。

```
$ python setup.py build_ext --inplace
```

代わりに、Cython コードを手動でコンパイルすることもできます。

```
$ cython multithreads.pyx
```

Python* 拡張コードを含む `multithreads.c` ファイルが生成されます。`gcc` コンパイラーで拡張コードをコンパイルすると、共有オブジェクト `multithreads.so` ファイルが生成されます。

```
$ gcc -O0 -shared -pthread -fPIC -fwrapv -Wall -fno-strict-aliasing
-fopenmp multithreads.c -I/opt/intel/intelpython27/include/python2.7
-L/opt/intel/intelpython27/lib -lpython2.7 -o multithreads.so
```

共有コードが生成されたら、このモジュールを Python* コードにインポートして、スレッド並列処理を有効にします。次のセクションでは、パフォーマンスを向上する方法を説明します。

timeit モジュールをインポートして Python* 関数の実行時間を測定することができます。デフォルトでは、timeit は関数を 1,000,000 回実行します。実行時間を短縮するため、次の例では実行回数を 100 に設定しています。基本的に、timeit.Timer() は multithreads モジュールをインポートして、multithreads.test_serial() 関数で費やされた時間を測定します。引数 number=100 は、100 回実行するように Python* インタープリターに伝えます。つまり、t1.timeit(number=100) は、シリアルループ (1 スレッドのみループを実行) を 100 回実行した場合の時間を測定します。

同様に、t12.timeit(number=100) は、並列ループ (複数のスレッドが並列で計算を実行) を 100 回実行した場合の時間を測定します。

- gcc コンパイラーで -O0 オプション (すべての最適化を無効にする) を指定してシリアルループを測定します。

```
$ python
Python 2.7.12 |Intel Corporation| (default, Oct 20 2016, 03:10:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Intel(R) Distribution for Python is brought to you by Intel Corporation.
Please check out: https://software.intel.com/en-us/python-distribution
```

timeit および t1 をインポートして、シリアルループで費やされた時間を測定します。gcc コンパイラーですべての最適化を無効にしてビルドすることに注意してください。結果は秒で表示されます。

```
>>> import timeit
>>> t1 = timeit.Timer("multithreads.test_serial()","import multithreads")
>>> t1.timeit(number=100)
2874.419779062271
```

- gcc コンパイラーで -O0 オプション (すべての最適化を無効にする) を指定して並列ループを測定します。

並列ループを t2 で測定します (gcc コンパイラーですべての最適化を無効にしてビルドします)。

```
>>> t2 = timeit.Timer("multithreads.test_parallel()","import multithreads")
>>> t2.timeit(number=100)
26.016316175460815
```

上記の測定結果から、並列ループではパフォーマンスが約 110 倍に向上していることが分かります。

- icc コンパイラーで -O0 オプション (すべての最適化を無効にする) を指定して並列ループを測定します。

次に、インテル® C コンパイラーを使用して再コンパイルし、パフォーマンスを比較します。インテル® C/C++ コンパイラーでは、-fopenmp の代わりに -qopenmp オプションを指定して OpenMP* を有効にします。インテル® Parallel Studio XE 2017 をインストールした後、適切な環境変数を設定して、以前のビルドをすべて削除します。

```
$ source /opt/intel/parallel_studio_xe_2017.1.043/psxevars.sh intel64
Intel(R) Parallel Studio XE 2017 Update 1 for Linux*
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
```

```
$ rm multithreads.so multithreads.c -r build
```

このアプリケーションを icc コンパイラーを使用して明示的にコンパイルするには、次のコマンドを使用して setup.py ファイルを実行します。

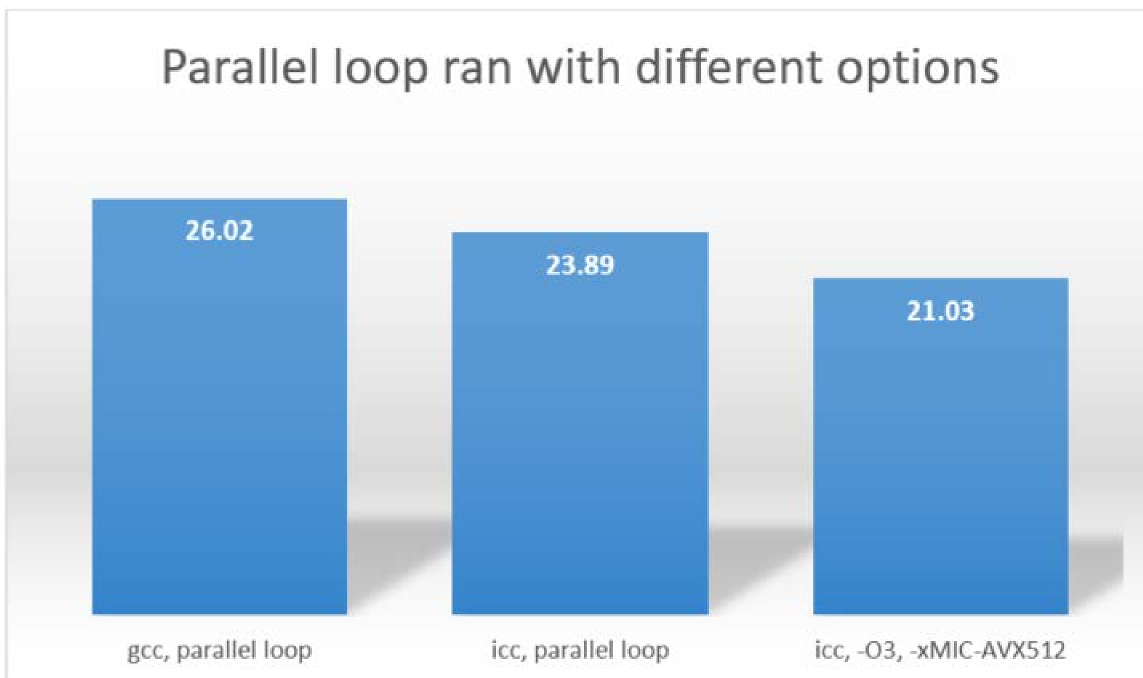
```
$ LDSHARED="icc -shared" CC=icc python setup.py build_ext --inplace
```

並列ループを t2 で測定します (今回は、インテル® コンパイラーですべての最適化を無効にしてビルドします)。

```
$ python
>>> import timeit
>>> t2 = timeit.Timer("multithreads.test_parallel()","import multithreads")
>>> t2.timeit(number=100)
23.89365792274475
```

- icc コンパイラーで -O3 オプションを指定して並列ループを測定します。

最後に、-O3 オプションを使用して、インテル® Xeon Phi™ プロセッサー向けにインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) ISA を有効にした場合のパフォーマンスを確認します。setup.py の -O0 を -O3 に変更して、-xMIC-AVX512 オプションを追加します。再コンパイルした後、前のステップで示されたように並列ループを実行します。結果は 21.027512073516846 になりました。次の図は、gcc で最適化を無効にした場合、icc で最適化を無効にした場合、icc で最適化、インテル® AVX-512 ISA を有効にした場合の結果 (単位は秒) を示しています。



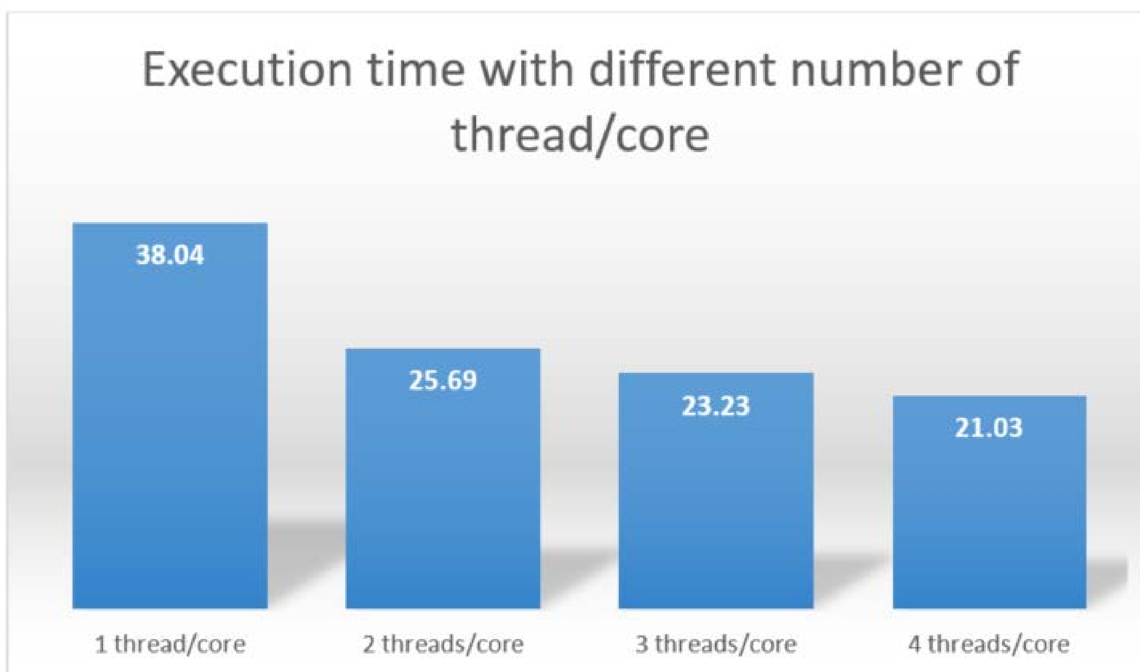

図から、インテル® コンパイラーを使用して、自動ベクトル化（-O3）およびインテル® Xeon Phi™ プロセッサー向けのインテル® AVX-512 ISA（-xMIC-AVX512）を有効にしてコンパイルした場合に最高の結果 (21.03 秒) が得られることが分かります。

デフォルトでは、インテル® Xeon Phi™ プロセッサーは利用可能なリソースをすべて使用し (68 コア、各コアで 4 つのハードウェア・スレッドを使用)、並列領域で合計 272 スレッドまたは 4 スレッド/コアを実行します。各コア

で実行するコア数とスレッド数は変更できます。最後のセクションでは、環境変数を使用してコア数とスレッド数を変更する方法を示します。

- 68 コアで 68 スレッド (コアあたり 1 スレッド) 使用してループ本体を 100 回実行するには、KMP_PLACE_THREADS 環境変数を次のように設定します。
  `$ export KMP_PLACE_THREADS=68c,1t`
- 68 コアで 136 スレッド (コアあたり 2 スレッド) 使用してループ本体を 100 回実行するには、KMP_PLACE_THREADS 環境変数を次のように設定します。
  `$ export KMP_PLACE_THREADS=68c,2t`
- 68 コアで 204 スレッド (コアあたり 3 スレッド) 使用してループ本体を 100 回実行するには、KMP_PLACE_THREADS 環境変数を次のように設定します。
  `$ export KMP_PLACE_THREADS=68c,3t`

次の図は結果をまとめたものです。




## まとめ

この記事では、Cython を利用して、インテル® Xeon Phi™ プロセッサーのマルチスレッド・サポートを活用する Python* 向けの拡張モジュールを作成する方法を説明しました。次に、セットアップ・スクリプトを使用して共有ライブラリーをビルドする方法を説明しました。並列ループのパフォーマンスは、セットアップ・スクリプトで異なるコンパイラー・オプションを指定することにより向上できます。この記事では、コアあたりのスレッド数を設定する方法についても説明しました。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。