インテル® Advisor Python* API を使用したパフォーマンス向上の考察

この記事は、Tech.Decoded に公開されている「Gaining Performance Insights Using the Intel® Advisor Python* API」の日本語参考訳です。

コードのチューニング方法を決定する適切なデータの取得

インテル コーポレーション、テクニカル・コンサルティング・エンジニア、Kevin O'Leary
インテル コーポレーション、シニア・ソフトウェア・デベロッパー、Egor Kazachkov

優れたデザインの決定には適切なデータが必要です。

- 最初にどのループをスレッド化およびベクトル化すべきか?
- 労力に見合うパフォーマンスの向上が得られるか?
- コア数が増えるとスレッド・パフォーマンスはスケーリングするか?
- このループにベクトル化を妨げる依存性があるか?
- トリップカウントとメモリー・アクセス・パターンは?
- 最新のインテル® 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) 命令を使用して効率的なベクトル化を行っているか? それとも古い SIMD 命令を使用しているか?

インテル® Advisor は、モダンなコードの構築と最適化を支援するインテルの包括的なツールスイート、**インテル® Parallel Studio XE** に含まれる動的解析ツールです。インテル® Advisor を使用すると、上記の質問の答えを得ることができます。アプリケーションのベクトル化およびメモリー・プロファイルに関する詳細なプログラムメトリックを収集できます。GUI やコマンドラインを使用してカスタマイズされたレポートを提供する従来の機能に加えて、Python* を使用して収集したデータベースから強力な新しいレポートを作成する機能が追加されました。

インテル® Advisor を実行すると、収集したすべてのデータは Python* API を使用してアクセスできる専用のデータベースに格納されます。この API を使用することにより、柔軟な方法でプログラムメトリックに関するカスタマイズされたレポートを生成できます。この記事では、この新しい機能の使用方法を説明します。

はじめに

最初に、インテル® Advisor 環境をセットアップします。(この記事では、スクリプトをすべて Linux* で実行しましたが、インテル® Advisor Python* API は Windows* もサポートしています。)

```
source advixe-vars.sh
```

次に、収集を行ってインテル® Advisor データをセットアップします。一部のプログラムメトリックは、トリップカウント、メモリー・アクセス・パターン、依存性などの追加の解析が必要です。

```
advixe-cl --collect survey --project-dir ./your_project -- <your-executable-
with-parameters>

advixe-cl --collect tripcounts -flops-and-masks -callstack-flops --project-
dir
 ./your_project -- <your-executable-with-parameters>
```

マップまたは依存性収集を行う場合は、解析するループを指定します。この情報は、インテル® Advisor GUI またはコマンドライン・レポートを使用して入手できます。

```
advixe-cl --collect map –mark-up-list=1,2,3,4 --project-dir ./your_project --
<your-executable-with-parameters>

advixe-cl --collect dependencies –mark-up-list=1,2,3,4 --project-
dir ./your_project --
<your-executable-with-parameters>
```

最後に、インテル® Advisor の参考例をテスト領域にコピーします。

```
cp -r /opt/intel/advisor_2018/pythonapi/examples .
```

この記事のスクリプトはすべて、インテル® Advisor Linux* 版に同梱されている Python* を使用して実行しました。標準の Python* ディストリビューションでも同様に動作するはずです。

インテル® Advisor Python* API

ここで提供した参考例は、この柔軟な方法を使用してプログラムデータにアクセスする小さなセットのレポートです。利用可能なフィールドのリストは columns.py を使用して取得できます。例えば、基本的な調査メトリックを実行すると、**表 1** のメトリックが表示されます。

Capability	Code
Compiler version	compiler_version
Vectorization status	is_vectorized
Mangled function or loop	mangled_name
Data types provided by binary static analysis	data_types
Loop unroll factor applied by compiler	unroll_factor

表 1. サンプル調査メトリック

インテル® Advisor Python* API の使用例

インテル® Advisor Python* API を使用して強力なメトリックを収集する方法を説明する単純な例から始めましょう。最初に、インテル® Advisor ライブラリー・パッケージをインポートします。

```
import advisor
```

収集した結果を含むインテル® Advisor プロジェクトを開きます。

```
project = advisor.open_project(sys.argv[1])
```

または、プロジェクトを作成して収集を実行します。(次の例では、open_project を実行しています。)この例では、メモリー・アクセス・パターン (MAP) 収集のデータにアクセスします。次のコード行を使用します。

```
data = project.load(advisor.MAP)
```

このデータをロードしたら、ループしてキャッシュ使用率の統計を収集します。次に、収集したデータを出力します。

```
import sys
try:
# First import the Advisor library
        import advisor
except ImportError:
        sys.exit(1)
# Open your Advisor project
        project = advisor.open_project(sys.argv[1])
```

```
# Load the Memory Access Pattern(MAP) data
        data = project.load(advisor.MAP)
#Loop through the MAP data and gather information about cache utilization
for site in data.map:
        site_id = site['site_id']
        cachesim = data.get_cachesim_info(site_id)
        print(indent * 2 + 'Average utilization'.ljust(width) + ' =
{:.2f}%'.format(cachesim.utilization))
```

インテル® Advisor Python* API の高度な機能

インテル® Advisor Python* API の一部として提供される例は、独自のスクリプトを記述する参考になります。**表 2** は、これらの高度な機能の一部です。

Capability	Code
Create an Intel Advisor project	`compiler_version project = advisor.create_project(project_dir)`
Run an Intel Advisor survey collection	`data = project.collect(advisor.SURVEY)`
Run an Intel Advisor tripcounts collection	`data = project.collect(advisor.TRIPCOUNTS`
Run an Intel Advisor tripcounts collection and also collect FLOPS data	`data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop'])`
Run an Intel Advisor tripcounts collection but only collect FLOPS and not tripcounts	`data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop', 'no-trip-counts'])`
Run an Intel Advisor Roofline collection	`data = project.collect(advisor.ROOFLINE)`
Run an Intel Advisor memory access pattern (MAP) collection	`data = project.collect(advisor.MAP)`
Run an Intel Advisor dependencies collection	`data = project.collect(advisor.DEPENENCIES)`

表 2. インテル® Advisor Python* API の高度な機能

いくつかの例を次に示します。例のリストは定期的に追加されています。

収集したすべてのデータを表示するレポートを生成します。

```
project = advisor.create_project(project_dir)
```

html レポートを生成します。

```
advixe-python to_html.py ./your_project
```

ルーフライン HTML グラフ **(図 1)** を作成します。

```
python roofline.py ./your_project
```

`roofline.py` スクリプトは、`advixe-python` **ではなく、外部 Python* コマンドで実行する必要があります。**現在は、Linux* でのみ動作します。追加のライブラリー (NumPy*、pandas、matplotlib) をインストールする必要もあります。キャッシュ・シミュレーション統計を生成します。

```
advixe-python cache.py ./your_project
```

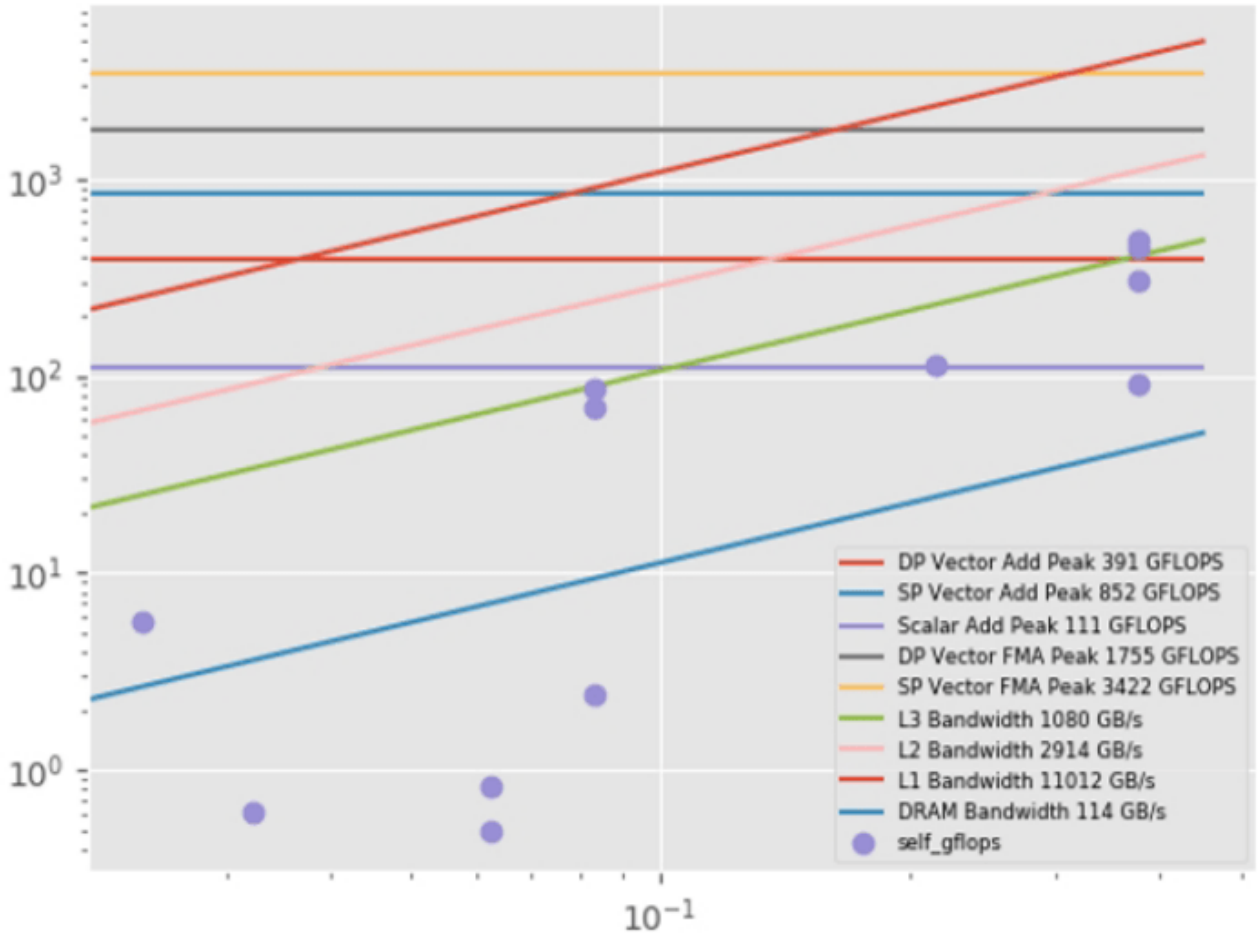

図 1 - ルーフライン HTML グラフ

キャッシュモデルから取得した結果を**表 3** に示します。

Capability	Result
Writes	46
Reads	92
Read misses	50
Average evicted cache line utilization	6.25%
Evicted cache line bytes used	4
Evicted lines	48

表 3. キャッシュモデルの結果

ケーススタディー: ベクトル化の比較

このケーススタディーでは、異なるコンパイラー・オプションを使用してコンパイルしたときに指定されたループのベクトル化を比較する Python* スクリプトを作成します。

ステップ 1: 異なる最適化オプションを使用してコードをコンパイルする

最初に、異なるオプションを指定してコードをコンパイルします。この例では**インテル® C++ コンパイラー** を使用しています (インテル® Advisor はバイナリーレベルで動作するため、任意のコンパイラーを使用できます)。最初のケースは、コンパイラー・オプション -O0 を使用して最適化なしでコンパイルしています。2 つ目のケースは、オプション -O3 を使用して最適化を行っています。

```
icc loops1.cpp -O0 -g -debug inline-debug-info -qopt-report=5 -ipo- -o
loops1-no-opt
icc loops1.cpp -O3 -g -debug inline-debug-info -qopt-report=5 -ipo- -o loops1
```

ステップ 2: Python* コード

スクリプトは非常に単純です。最初に、コマンドラインからいくつかの引数を取得します。引数がインテル® Advisor プロジェクトに渡されている場合は、プロジェクトに含まれていたデータを使用します。そうでない場合は、インテル® Advisor の調査を実行します。調査の実行が完了したら、ループのアセンブリーをデコードし

て、2 つのループの命令を並べて出力します。Python* コードのメイン関数は get_formatted_asm です。この関数は、インテル® Advisor データベースにアクセスしてループのアセンブリーをデコードします。アセンブリー・コードがベクトル命令を使用しているかどうか、ループがどの程度速く実行されているかもチェックします。

```
import sys
import itertools

import advisor

# second form allows collecting data before analysis
# first form just analyses already collected data

if len(sys.argv) < 3 or len(sys.argv) > 6:
        print('''

Usage:
advixe-python {} path_to_project_dir loop1 [loop2]

 Or:
advixe-python {} path_to_project_dir loop1 [loop2] executable1 executable2

Where loop1 and loop2 are in the form source:line

'''.format(__file__, __file__))
    sys.exit(1)

project_dir = sys.argv[1]
project_dir1 = project_dir + ".1"
project_dir2 = project_dir + ".2"

loop1 = sys.argv[2]
# if we have an odd number of arguments (including script name) then loop2 is the
same as loop1
loop2 = sys.argv[3] if len(sys.argv)%2 == 0 else loop1

binary1 = ''
binary2 = ''

# in the second form two last args are executables to run
if 4 < len(sys.argv) < 7:
binary1 = sys.argv[-2]
binary2 = sys.argv[-1]

# try open or create project, run collection if needed:
# returns formatted asm listing with vectorized instructions marked with "VEC "
for given loop
def get_formatted_asm(project_dir, binary, loop):
    asm = []

    try:
        project = advisor.open_project(project_dir)
    except:
        project = advisor.create_project(project_dir)

        if binary:
            project.collect(advisor.SURVEY, binary)

        data = project.load(advisor.SURVEY)
```

```
        for entry in data.bottomup:
            if loop in entry['function_call_sites_and_loops']:
              asm += ["{:54.54} ".format(entry['function_call_sites_and_loops']),
                                "{:54.54} ".format("Self time: " +
entry['self_time']),
                                " "*54]
                for instruction in entry.assembly:
                    isVectorized = "VEC" if "VECTORIZED" in
instruction['instruction_type'] else ""
                    asm.append("{:4.4}{:50.50} ".format(isVectorized,
instruction['asm']))
                asm.append("")
        return asm

asm1 = get_formatted_asm(project_dir1, binary1, loop1)
asm2 = get_formatted_asm(project_dir2, binary2, loop2)

# print alongside asm listings for comparison
for (a1,a2) in itertools.izip_longest(asm1, asm2, fillvalue = ' '*40):
    print("{}{}".format(a1,a2))
```

ステップ 3: Python* スクリプトを実行する

```
advixe-python compare_asm.py /home/work/projects/loops-compare loops1.cpp:34
/home/work/tests/loops/loops1-no-opt /home/work/tests/loops/loops1
[loop in main at loops1.cpp:34]
Self time: 45.5463

Block 1
 movl  -0xbc(%rbp), %eax
 movsxd %eax, %rax
 imul $0x8, %rax, %rax
 addq  -0x88(%rbp), %rax
 movl  -0xac(%rbp), %edx
 imull  -0xac(%rbp), %edx
 mov $0x1, %ecx
 addl  -0xac(%rbp), %ecx
 movq  %rax, -0x28(%rbp)
 mov %edx, %eax
 cdq
 idiv %ecx
 cvtsi2sd %eax, %xmm0
 movl  -0xc0(%rbp), %eax
 cvtsi2sd %eax, %xmm1
 movsdq  0x555(%rip), %xmm2
 divsd %xmm2, %xmm1
 addsd %xmm1, %xmm0
 movq  -0x28(%rbp), %rax
 movsdq  (%rax), %xmm1
 subsd %xmm0, %xmm1
 movl  -0xbc(%rbp), %eax
 movsxd %eax, %rax
 imul $0x8, %rax, %rax
 addq  -0x88(%rbp), %rax
 movsdq  %xmm1, (%rax)
 mov $0x1, %eax
 addl  -0xac(%rbp), %eax
 movl  %eax, -0xac(%rbp)
```

```
 movl  -0xac(%rbp), %eax
 cmp $0x64, %eax
 jl 0x401020 <Block 1>
[loop in main at loops1.cpp:34]
Self time: 4.62404

      Block 1
  VEC movdqa %xmm11, %xmm2
  VEC movdqa %xmm11, %xmm0
  VEC psrlq $0x20, %xmm2
  VEC movdqa %xmm10, %xmm1
  VEC pmuludq %xmm2, %xmm2
  VEC pmuludq %xmm11, %xmm0
  VEC psllq $0x20, %xmm2
  VEC pand %xmm13, %xmm0
  VEC por %xmm2, %xmm0
  callq  0x401770 <__svml_idiv4>
      Block 2
  VEC cvtdq2pd %xmm0, %xmm2
  VEC punpckhqdq %xmm0, %xmm0
      add $0x4, %r15b
  VEC cvtdq2pd %xmm0, %xmm3
  VEC addpd %xmm14, %xmm2
  VEC addpd %xmm14, %xmm3
  VEC subpd %xmm2, %xmm12
  VEC subpd %xmm3, %xmm8
  VEC paddd %xmm15, %xmm11
  VEC paddd %xmm15, %xmm10
      cmp $0x64, %r15b
      jb 0x401397 <Block 1>
```

ステップ 4: オプション -AVX2 を追加して再コンパイルする

もう少し最適化してみましょう。プロセッサーはインテル® AVX2 命令セットをサポートしているため、AVX2 向けのコードを生成するようにコンパイラーに指示します。(コンパイラーはデフォルトでこのコードを生成しないことに注意してください。)

```
icc loops1.cpp -O3 -xCORE-AVX2 -g -debug inline-debug-info -qopt-report=5 -
ipo- -o loops1-avx2
```

ステップ 5: 比較を再実行する

```
advixe-python compare_asm.py /home/work/projects/loops-compare-opt
loops1.cpp:34
/home/work/tests/loops/loops1 /home/work/tests/loops/loops1-avx2
[loop in main at loops1.cpp:34]
Self time: 4.81401
    Block 1
VEC movdqa %xmm11, %xmm2
VEC movdqa %xmm11, %xmm0
VEC psrlq $0x20, %xmm2
VEC movdqa %xmm10, %xmm1
VEC pmuludq %xmm2, %xmm2
VEC pmuludq %xmm11, %xmm0
```

```
VEC psllq $0x20, %xmm2
VEC pand %xmm13, %xmm0
VEC por %xmm2, %xmm0
    callq 0x401770 <__svml_idiv4>
    Block 2
VEC cvtdq2pd %xmm0, %xmm2
VEC punpckhqdq %xmm0, %xmm0
    add $0x4, %r15b
VEC cvtdq2pd %xmm0, %xmm3 cmp $0x60, %r15b
VEC addpd %xmm14, %xmm2 jb 0x4015a9
VEC addpd %xmm14, %xmm3
VEC subpd %xmm2, %xmm12
VEC subpd %xmm3, %xmm8
VEC paddd %xmm15, %xmm11
VEC paddd %xmm15, %xmm10
    cmp $0x64, %r15b
    jb 0x401397 <Block 1>
[loop in main at loops1.cpp:34]
Self time: 1.97998
       Block 1
VEC vpmulld %ymm15, %ymm15, %ymm0
   VEC vmovdqa %ymm14, %ymm1
  callq 0x4018 f0<__svml_idiv8>
      Block 2
      add $0x8, %r15b
VEC vextracti128 $0x1, %ymm0, %xmm2
    VEC vcvtdq2pd %xmm0, %ymm3
VEC vpaddd %ymm13, %ymm15, %ymm15
VEC vcvtdq2pd %xmm2, %ymm5
VEC vpaddd %ymm13, %ymm14, %ymm14
VEC vaddpd %ymm3, %ymm10, %ymm4
VEC vaddpd %ymm5, %ymm10, %ymm6
VEC vsubpd %ymm4, %ymm8, %ymm8
VEC vsubpd %ymm6, %ymm9, %ymm9
        cmp $0x60, %r15b
        jb 0x4015a9 <Block 1>
```

アセンブリー・コードで XMM の代わりにベクトル長が 2 倍の YMM レジスターが使用され、スピードが 2 倍になりました。

結果

最適化および最新のベクトル命令セットを使用したことにより、実行時間は大幅に短縮されました。

- **最適化なし (-O0):** 45.148 秒
- **最適化 (-O3):** 4.403 秒
- **最適化およびベクトル化 (-O3 -AVX2):** 2.056 秒

システム・パフォーマンスの最大化

最近のプロセッサーのパフォーマンスを最大限に引き出すには、ソフトウェアをベクトル化およびスレッド化することが重要です。インテル® Parallel Studio XE の新しいインテル® Advisor Python* API は、システムのパフォーマンスを最大限に引き出すために役立つプログラム統計とレポートを生成する強力な方法を提供します。この記事で説明した例は、この新しいインターフェイスの能力を示しています。各自のニーズに基づいて、これらの例は調整および拡張できます。インテルでは、インテル® Advisor Python* API についてのフィードバックを受け付けています。ご意見、フィードバックは、**mvector_advisor@intel.com** までメール (英語) でご連絡ください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、http://www.intel.com/performance (英語) を参照してください。

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

#DataScience (英語)

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。