Docker*、WSL、oneAPI: コンテナー化されたワークロードを最適化する方法

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「Docker, WSL, and oneAPI — A Quick How-To Guide」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

以前の記事では、開発者にとって効率的であることの重要性について説明しました。コンテナーの普及は、この目標達成に大いに役立ちます。

コンテナーとは?

最初に、コンテナーとは何かについて簡単に説明したいと思います。以下は、最も一般的なコンテナー・ソリューションである Docker* のウェブサイト (英語) から引用したコンテナーの定義です。

> コンテナーとは、コードとそのすべての依存関係をパッケージ化したソフトウェアの標準単位であり、アプリケーションをあるコンピューティング環境から別の環境へ移行しても、迅速かつ確実に実行できるようにします。Docker* コンテナーのイメージは、アプリケーションの実行に必要なすべてのもの (コード、ランタイム、システムツール、システム・ライブラリー、設定) を含む、軽量でスタンドアロンの実行可能なソフトウェア・パッケージです。

コンテナーは Windows* や Linux* システム上で動作し、セットアップや展開が比較的簡単で、(Docker* Hub (英語) などのサービスを介して) ほかの開発者と共有したり配布するのも簡単です。

コンテナーには、次のような制限があります。

- コンテナーを実行するすべてのシステムでコンテナーランタイムをセットアップする必要があります。
- コンテナーは非常に大きくなる可能性があり、頻繁にアップロードやダウンロードを行う必要がある場合には問題となります。
- 仮想マシン (VM) のように、ホストシステム上のリソースをハードウェア・レベルで分離できません。

これらの制限にもかかわらず、多くのユースケースにおいて、コンテナーは素晴らしい方法であり、開発者にとってどのように役立つかを説明します。

開発者にとってのメリット

開発プロセスの一部としてコンテナーを使用することで、安定した再現可能なソフトウェア開発環境を作成し、維持することが可能です。開発環境が壊れても、コンテナーの別のインスタンスを起動するだけで、簡単にクリーンな環境で作業できます。

また、1 つのシステムで複数の開発環境とテスト環境 (複数の Linux* バージョン) を作成し、実行することも可能です。最も重要なことは、コンテナーがほとんどのリソースを分離するので、一般にホストシステムに影響を与える心配がないことです。

顧客がコンテナーを使用している場合、コンテナーでアプリケーションを配布することで、より制御された環境で実行することができます。これにより、不具合を減らし、テスト領域を縮小し、製品化までの期間 (TTM) を短縮することができ、開発者と顧客の双方にメリットをもたらします。

仮想マシン (VM) ベースのソリューションなど、「複数の開発環境」の課題を解決するオプションはほかにもあります。Windows* Subsystem for Linux* (WSL) インスタンスを作成し、その中で開発環境を構成し、そのインスタンスを複数回デプロイすることもできます。また、VMware* 仮想デスクトップ・インフラストラクチャ (VDI) のような、より堅牢なエンタープライズ向けソリューションを利用することも可能です。しかし、これらのソリューションは、すべてのプラットフォームで誰もが利用できるわけではありません。そこで、このブログでは、oneAPI を使用したソフトウェアの開発と展開を支援するコンテナーの活用に注目します。

oneAPI 開発用コンテナーの利用

oneAPI 開発用コンテナーを利用して、前述のような安定した開発環境を提供するには、まず開発システムで Docker* をセットアップする必要があります。Docker* のウェブサイトにアクセスして、インストール手順に従います。以前の記事で触れたように、WSL 環境で開発している場合は追加の作業が必要になります。

このことについては、すでに多くの解説がありますが、ferarias (英語) に掲載されているこちらの記事 (英語) が分かりやすいでしょう。こちらの日本語記事も役立ちます。

Docker* をセットアップしたら、Docker* Hub から最新の oneAPI 開発用コンテナーを取得します。

CentOS* 8、Ubuntu* 18.04、Ubuntu* 20.04 用のコンテナーがありますが、この例では簡単な SYCL* コードをビルドしてパッケージ化するので、oneAPI ベース・ツールキットの Ubuntu* 20.04 イメージをプルします。Docker* のコマンドラインで以下を実行します。

```
> docker pull intel/oneapi-basekit:devel-ubuntu20.04
```

これで、コンテナーがローカルシステムにダウンロードされます。開発環境に入るには、以下のコマンドでコンテナーを実行します。

```
> docker run -ti --name=ubuntu-dev-20.04 intel/oneapi-basekit:devel-ubuntu20.04
```

`-ti` フラグは、コンテナーが起動したら、コンテナー内で対話型端末を提供するように Docker* に指示します。name フラグは、実行中のコンテナーに ubuntu-dev-20.04 という名前を付けます。

コードのビルド

説明を簡潔にするため、oneAPI のサンプルを使用します。GitHub* の oneAPI サンプル (英語) にアクセスします。

そして、このリポジトリーを開発用コンテナーにクローンします。ここでは、`DirectProgramming->DPC++->N-BodyMethods->Nbody` フォルダーにある `Nbody` サンプルを使用します。

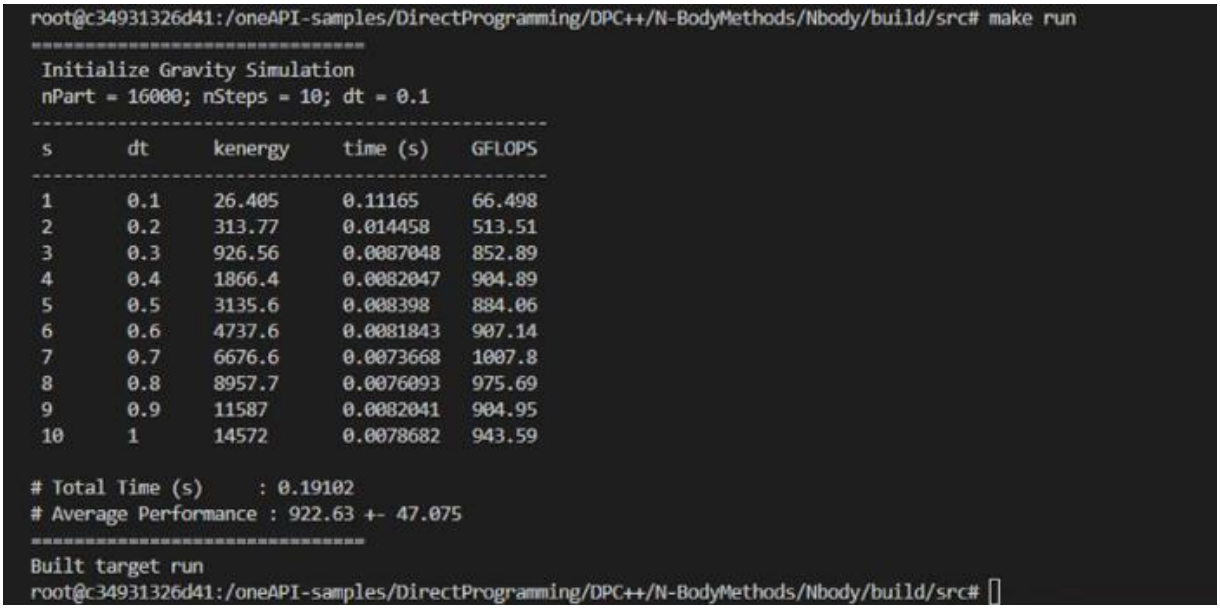
開発用コンテナーでは、以下の環境変数を設定するスクリプトを実行する必要がありません。

```
> source /opt/intel/oneapi/setvars.sh
```

oneAPI 開発用コンテナーは上記のコマンドを実行した状態でビルドされています。そのため、`Nbody` フォルダーに移動して、ビルド手順に従って `Nbody` サンプルをビルドして実行することができます。

```
> mkdir build
> cd build
> cmake ..
> make
> make run
```

以下は、インテル® Core™ i9 プロセッサー (開発コード名 Alder Lake) 搭載の Alienware* R13 システムで実行した結果です。



本番用コンテナーの作成

開発用コンテナーでコードをビルドできたので、顧客のシステム上で実行可能なコンテナーとして配布する方法を見てみましょう。

Docker* は、Dockerfile (英語) 形式を使用して新しいコンテナーを構築します。Dockerfile の使い方は、Docker* の導入ガイド (英語) を参照してください。

Docker* のマルチステージ・ビルド機能 (英語) を利用して、Docker* のビルドプロセスに、コードをあるコンテナーでビルドし、それを別のコンテナーにコピーするように指示します。これにより、ビルドとパッケージのワークフロー全体を 1 カ所で行い、開発ツールを含まないコンテナーを配布することができます。以下は、本番用コンテナーの Dockerfile です。

```
# run the development container and name it mybuild
FROM intel/oneapi-basekit:devel-ubuntu20.04 as mybuild

# get oneAPI sample code
RUN git clone https://github.com/oneapi-src/oneAPI-samples

# build the Nbody sample to root directory
RUN cmake /oneAPI-samples/DirectProgramming/DPC++/N-BodyMethods/Nbody
RUN make

# use oneapi-runtime container as my production container
FROM intel/oneapi-runtime:latest

# copy file from mybuild to the production container
COPY - from=mybuild src/nbody /
CMD ["/nbody"]
```

本番用コンテナーは、oneAPI ベースのワークロードの実行に必要なすべてのランタイムを含む `intel/oneapi-runtime:latest` コンテナーをベースにしていることが分かります。

次に、この Dockerfile (`Dockerfile.runtime`) を使用して、docker build コマンドを実行し、新しいコンテナーを作成します。

```
> docker build . -f Dockerfile.runtime -t tonymintel/nbody:runtime
```

このコマンドは、現在のパスで `Dockerfile.runtime` ファイルを使用してビルドし、`tonym/nbody:runtime` としてタグ付けするように、Docker* に指示します。

```
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /opt/intel/oneapi/compiler/2022.1.0/linux/bin/dpcpp - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /
Removing intermediate container f3eaf1e3be1a
 ---> 1bebc7895928
Step 4/7 : RUN make
 ---> Running in 2579020b33b4
[ 33%] Building CXX object src/CMakeFiles/nbody.dir/GSimulation.cpp.o
[ 66%] Building CXX object src/CMakeFiles/nbody.dir/main.cpp.o
[100%] Linking CXX executable nbody
[100%] Built target nbody
Removing intermediate container 2579020b33b4
 ---> c20af92acf3f
Step 5/7 : FROM intel/oneapi-runtime:latest
 ---> cde4d214d416
Step 6/7 : COPY --from=mybuild src/nbody /
 ---> 9ed7439456e8
Step 7/7 : CMD ["/nbody"]
 ---> Running in 0c3161dbaaaf
Removing intermediate container 0c3161dbaaaf
 ---> 2f8ebf85a86e
Successfully built 2f8ebf85a86e
Successfully tagged tonym/nbody:runtime
etmongko@AlienwareR13:~/dev/oneAPI-samples/DirectProgramming/DPC++/N-BodyMethods$
```

正常にビルドされました!

コンテナーを実行すると、正常にビルドされ、期待通りにコードが実行されていることが分かります。

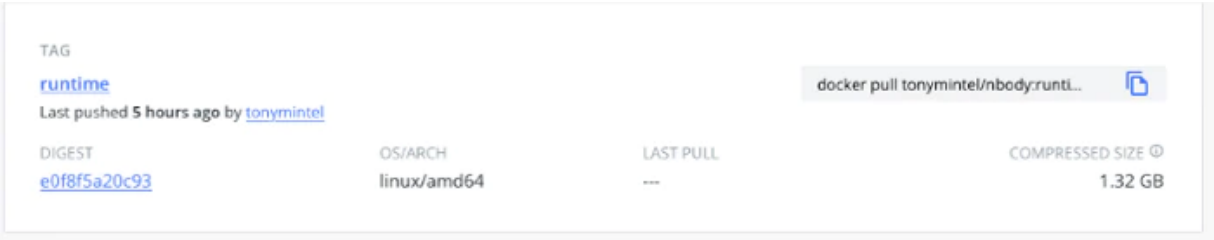
```
etmongko@AlienwareR13:~/dev/oneAPI-samples/DirectProgramming/DPC++/N-BodyMethods$ docker run tonymintel/nbody:runtime
================================
 Initialize Gravity Simulation
 nPart = 16000; nSteps = 10; dt = 0.1
------------------------------------------------
 s       dt      kenergy     time (s)    GFLOPS
------------------------------------------------
 1       0.1     26.405      0.27883     26.627
 2       0.2     313.77      0.011287    657.79
 3       0.3     926.56      0.0085317   870.21
 4       0.4     1866.4      0.010989    675.64
 5       0.5     3135.6      0.0080754   919.38
 6       0.6     4737.6      0.0082119   904.09
 7       0.7     6676.6      0.0076039   976.38
 8       0.8     8957.7      0.0074251   999.9
 9       0.9     11587       0.0077278   960.72
 10      1       14572       0.0073386   1011.7

# Total Time (s)     : 0.35611
# Average Performance : 914.75 +- 101.15
================================
etmongko@AlienwareR13:~/dev/oneAPI-samples/DirectProgramming/DPC++/N-BodyMethods$
```

あとは、コンテナーをパッケージ化して、顧客に配布するだけです。Docker* Hub にプッシュするか、docker save コマンドを使ってローカルシステムにイメージを保存し、そのファイルを配布できます。

ここでは、以下のコマンドで Docker* Hub にプッシュします。

```
> docker push tonymintel/nbody:runtime
```

イメージが Docker* Hub アカウント (この例では `tonymintel`) にプッシュされ、`nbody` リポジトリーに runtime タグ付きで保存されます。これで、Docker* Hub リポジトリー `tonymintel/nbody` にアクセスできる人なら誰でも、通常の Docker* コマンドでコードを取得して実行できます。



Docker* Hub のタグ付きの nbody リポジトリー

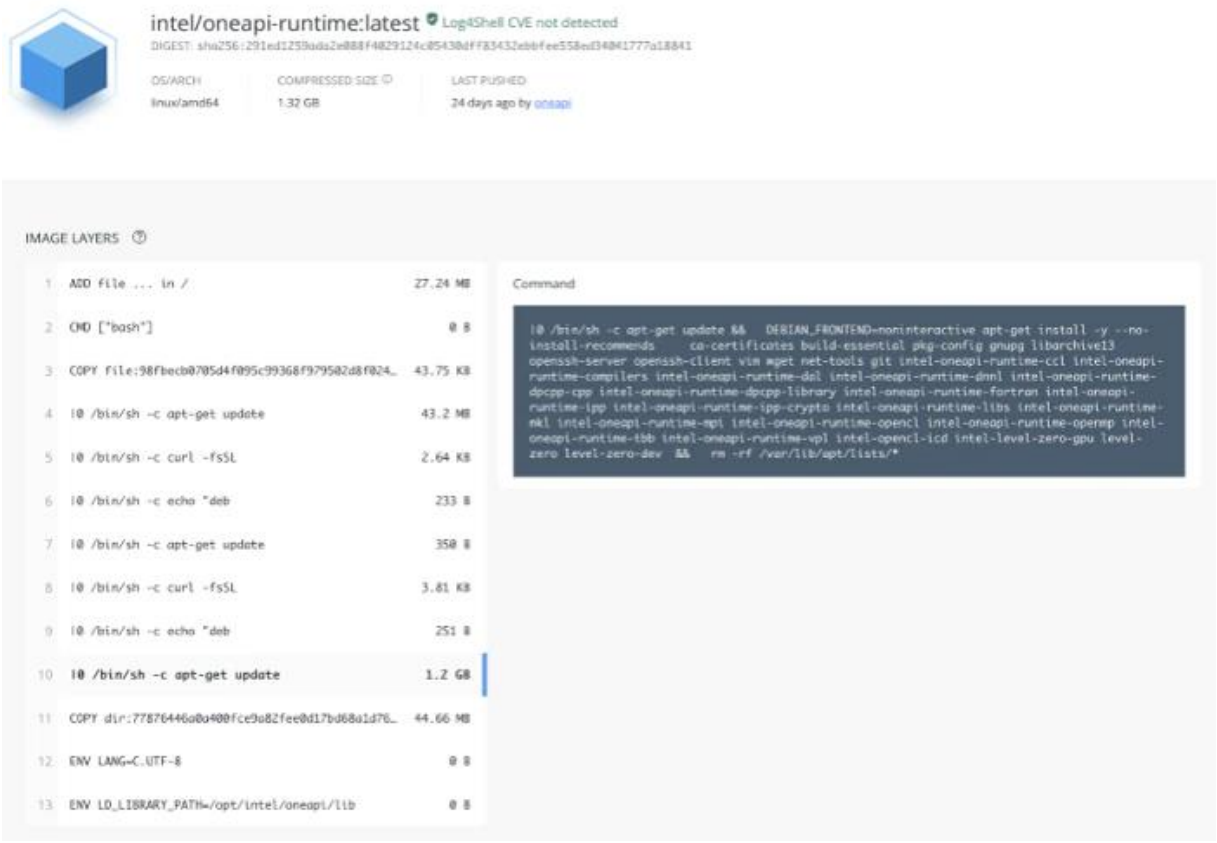
本番用コンテナーを 1/4 にする

開発者として効率的であることが重要であるように、アプリケーションのユーザーにとって効率的なソリューションは重要です。インテル社の James Reinders が言うように、「Joy Out of the Box (すぐに使える喜び)」を確実に提供する必要があります。

上記のスクリーンショットから、アップロードしたコンテナーは 1.32GB であることが分かります。これは、顧客がダウンロードするにはかなり大きな Docker* コンテナーです。

本番用コンテナーは、intel/oneapi-runtime コンテナーをベースにしており、oneAPI コードのビルドに使用するすべてのツールキットのランタイムが含まれています。これは必ずしも悪いことでありません。oneAPI はさまざまなライブラリーを提供しており、今後コードで使用するかもしれないランタイムパッケージを含めてお

くのは良いことでしょう。しかし、どのライブラリーを使用しているか知っている開発者として、本番用コンテナーを最適化して小さくすることができるか見てみましょう。



Docker Hub* の intel/oneapi-runtime:latest ファイル (2022 年 6 月現在)

色々と調べた結果、以下のような新しい Dockerfile.prod ファイルを作成しました。

```
# run the development container and name it mybuild
FROM intel/oneapi-basekit:devel-ubuntu20.04 as mybuild

# get oneAPI sample code
RUN git clone https://github.com/oneapi-src/oneAPI-samples

# build the Nbody sample to root directory
RUN cmake /oneAPI-samples/DirectProgramming/DPC++/N-BodyMethods/Nbody
RUN make

# build the production container
FROM ubuntu:20.04
RUN apt-get update && \
 DEBIAN_FRONTEND=noninteractive apt-get install -y - no-install-recommends \
 curl ca-certificates gpg-agent software-properties-common

# repository to install Intel(R) oneAPI Libraries
RUN curl -fsSL https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-
PRODUCTS-2023.PUB | apt-key add -
RUN echo "deb [trusted=yes] https://apt.repos.intel.com/oneapi all main " >
/etc/apt/sources.list.d/oneAPI.list

# repository to install Intel(R) GPU drivers
RUN curl -fsSL https://repositories.intel.com/graphics/intel-graphics.key | apt-
key add -
RUN echo "deb [trusted=yes arch=amd64]
https://repositories.intel.com/graphics/ubuntu focal main" >
/etc/apt/sources.list.d/intel-graphics.list
```

```
# pull required Intel runtime packages needed
RUN apt-get update && \
 DEBIAN_FRONTEND=noninteractive apt-get install -y - no-install-recommends \
 intel-oneapi-runtime-dpcpp-cpp
ENV LANG=C.UTF-8
ENV LD_LIBRARY_PATH=/opt/intel/oneapi/lib

# copy file from build container to our production container
COPY - from=build src/nbody /
CMD ["/nbody"]
```

詳しく見ていきましょう。

行 1 ~ 9 と 32 ~ 34 は、どちらの Dockerfile でも同じです。バイナリーをビルドして、コンテナーの開始時に実行します。

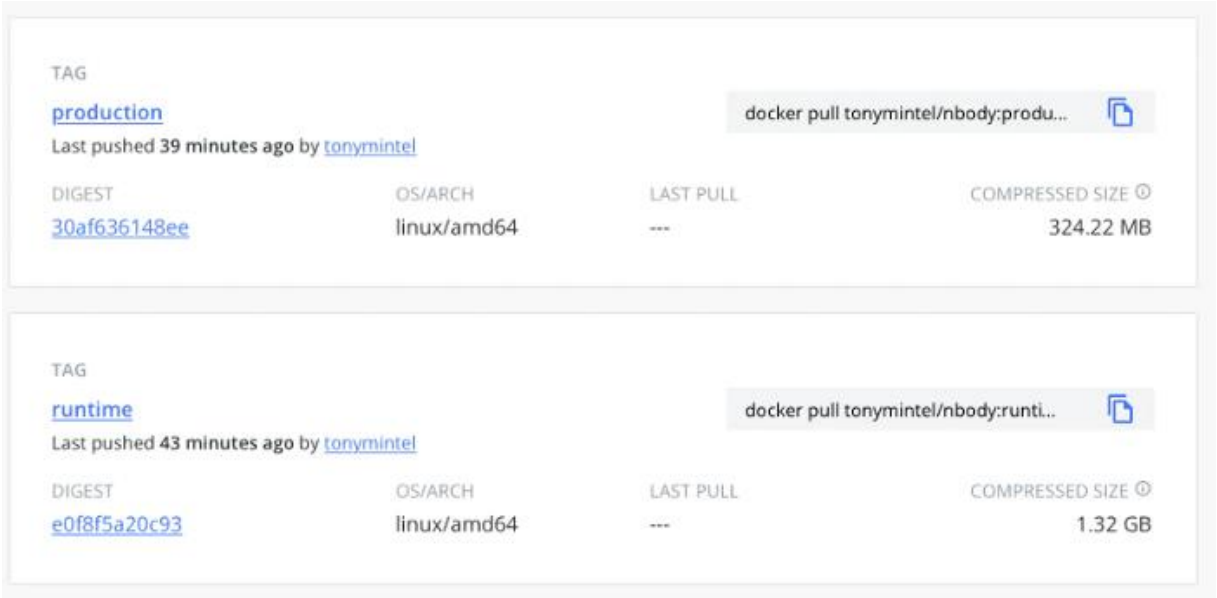
行 11 ~ 30 のほとんどは、インテルの apt リポジトリーをセットアップし、APT 経由でいくつかのインテルのパッケージをインストールする定型的なコードです。これらの行は、https://github.com/intel/oneapi-containers (英語) にある `oneapi-runtime` Dockerfile からプルされます。

`Dockerfile.prod` で重要なのは行 28 です。オリジナルの `intel/oneapi-runtime` Dockerfile (ここをクリック (英語) して、DIGEST にある SHA をクリックし、最大のイメージを見つけます) を確認すると、利用可能なすべてのランタイムパッケージの apt-get が行われていることが分かります。この例では、DPC++ のみを使用しているため、`Dockerfile.prod` には `intel-oneapi-runtime-dpcpp-cpp` パッケージのみを含めます。

新しい Dockerfile をビルドして、以前と同じように動作することを確認し、クラウドにプッシュします。

```
> docker build . -f Dockerfile.prod -t tonymintel/nbody:production
> docker run -it tonymintel/nbody:production
> docker push tonymintel/nbody:production
```

これで、顧客のダウンロード・サイズが約 1/4 になり、「Joy Out of the Box (すぐに使える喜び)」が向上しました。



さらに改良できないか、と思われるかもしれません。しかし、そのためには、ファイルを 1 つずつ確認して、未使用のランタイム・コンポーネントを排除する必要があります。これは少し面倒ですが、可能です。その結果、パッケージは約 250 MB (1/5) になりました。

これは、顧客体験の大幅な向上につながります。顧客が標準的なインターネット接続を介してコンテナーをダウンロードする必要がある場合に大きな利点となります。データセンターやクラウド環境などの大規模な環境では、同じポッドを数十または数百のノードに展開するのが一般的で、ネットワークの使用量、飽和、遅延が常に懸念されるため、さらに大きな価値をもたらすでしょう。さらに、コンテナーの展開にネットワークやストレージ費用が発生する場合、小さなコンテナーは費用軽減につながり喜ばれるでしょう。

まとめ

コンテナーは、開発者がコードを作成し、テストするのを助ける便利な技術です。コンテナーを使用する顧客には、より信頼性の高いコードを配布するのにも役立ちます。

技術の習得には多少の労力がつきものですが、多くの場合、それは間違いなく価値があります。コンテナーの活用方法と、顧客向けにコンテナーを最適化する方法について、少しはご理解いただけたのであれば幸いです。

また次回お会いしましょう。

著者について

著者が普段どのような技術ニュースをチェックしているか興味のある方は、著者の Twitter* アカウントをフォロー (英語) してください。

Tony Mongkolsmai は、インテル コーポレーションのソフトウェア・アーキテクト兼テクニカル・エバンジェリストです。いくつかのソフトウェア開発ツールの開発に携わってきました。最近では、Habana Labs のスケーラブルな MLPerf ソリューションを可能にするデータセンター・プラットフォームを構築したソフトウェア・エンジニアリング・チームを率いました。

関連情報

ポッドキャスト

- オープンソース・コラボレーションによるヘテロジニアスな未来の構築 (英語)

技術記事

- SYCL* と oneAPI でベンダーへの依存からソフトウェアを解放
- oneAPI 研究拠点でオープンな開発を加速 (英語)
- ヘテロジニアス並列処理のオープン・スタンダードベースのエコシステムの構築
- SYCLomatic: 新しい CUDA* から SYCL* へのコード移行ツール

ソフトウェアを入手

インテル® oneAPI ベース・ツールキット
ハイパフォーマンスなデータセントリックのアプリケーションを多様なアーキテクチャー向けに開発してデプロイするのに必要なツールとライブラリーの基本セットを提供します。

今すぐ入手 (英語)
すべてのツールを見る (英語)