Akceleracja obliczeń - standard SYCL

Błażej Smorawski

27.05.2023

Wstęp

Wraz z postępem technologicznym i coraz większą dostępnością danych, zapotrzebowanie na złożone obliczenia stale rośnie. W dzisiejszym środowisku, które jest napędzane przez rozwój sztucznej inteligencji, uczenia maszynowego, analizy danych i innych zaawansowanych technologii, obliczenia stają się nieodzownym narzędziem do przetwarzania ogromnych ilości informacji w sposób efektywny.

Tradycyjne jednostki centralne (*CPU*) osiągają swoje granice wydajności w obliczeniach równoległych, a tym samym nie są w stanie sprostać rosnącemu zapotrzebowaniu na szybkie przetwarzanie danych. W odpowiedzi na to wyzwanie, technologia *GPU* (Graphics Processing Unit) zyskuje coraz większą popularność jako narzędzie do akceleracji obliczeń. *GPU*, początkowo używane do przetwarzania grafiki komputerowej, okazują się być niezwykle skutecznymi w przetwarzaniu równoległym. Dzięki dużej liczbie rdzeni obliczeniowych *GPU* umożliwiają równoczesne wykonywanie wielu operacji na danych, co przekłada się na znaczne przyspieszenie czasu obliczeń.

Wraz z rozwojem sztucznej inteligencji, uczenia maszynowego i innych technologii opartych na danych, takich jak analiza danych czy symulacje naukowe, zapotrzebowanie na obliczenia staje się coraz bardziej złożone. Algorytmy uczenia maszynowego, które wymagają dużej mocy obliczeniowej, aby trenować modele na ogromnych zbiorach danych, są idealnym przykładem rosnącego zapotrzebowania na obliczenia. Ponadto, symulacje naukowe i analiza danych, które angażują coraz większą ilość informacji, wymagają wydajnych i skalowalnych rozwiązań obliczeniowych, aby generować dokładne wyniki w rozsądnym czasie.

W tym kontekście, skalowanie obliczeń na klastry obliczeniowe staje się nieodzowne dla efektywnego przetwarzania dużych obliczeń. Klastry obliczeniowe są zbiorem połączonych ze sobą węzłów obliczeniowych, które mogą równocześnie wykonywać zadania. Przez podział obliczeń na mniejsze części i równoległe przetwarzanie na różnych węzłach, możliwe jest zwiększenie wydajności i skrócenie czasu przetwarzania.

Czym jest SYCL?

SYCL jest jednym z rozwiązań, które możemy wykorzystać do akceleracji naszych obliczeń.

SYCL definiuje abstrakcje umożliwiające programowanie różnych akceleratorów, ważną zdolność we współczesnym świecie, która nie została jeszcze rozwiązana bezpośrednio w ISO C++. *SYCL* ewoluował z zamiarem wpłynięcia na kierunek C++ w zakresie heterogenicznych obliczeń.

Głównym celem *SYCL* jest umożliwienie korzystania z różnych urządzeń heterogenicznych w jednej aplikacji - na przykład jednoczesne korzystanie z procesorów *CPU*, *GPU* i *FPGA*. Chociaż zoptymal-

izowany kod kernela może różnić się w zależności od architektury (ponieważ *SYC*L nie gwarantuje automatycznego i doskonałego przenoszenia wydajności między architekturami), zapewnia spójny język, interfejsy API i ekosystem, w którym można pisać i dostrajać kod dla różnych architektur akceleratorów. Aplikacja może spójnie definiować warianty kodu zoptymalizowane pod kątem interesujących ją architektur, a także znajdować i wysyłać kod do tych architektur.

SYCL rewolucjonizuje tworzenie oprogramowania aplikacji wyższego poziomu, korzystając z potęgi standardowego programowania opartego na szablonach i funkcjach lambda. Dzięki temu, pisanie kodu staje się niezwykle klarowne, umożliwiając jednocześnie zoptymalizowane generowanie kodu kerneli obliczeniowych. *SYCL* współpracuje sprawnie z różnymi interfejsami API, takimi jak *OpenCL*, *CUDA* i *level-zero*, poszerzając możliwości programowania i otwierając drzwi do szerokiego spektrum zastosowań.

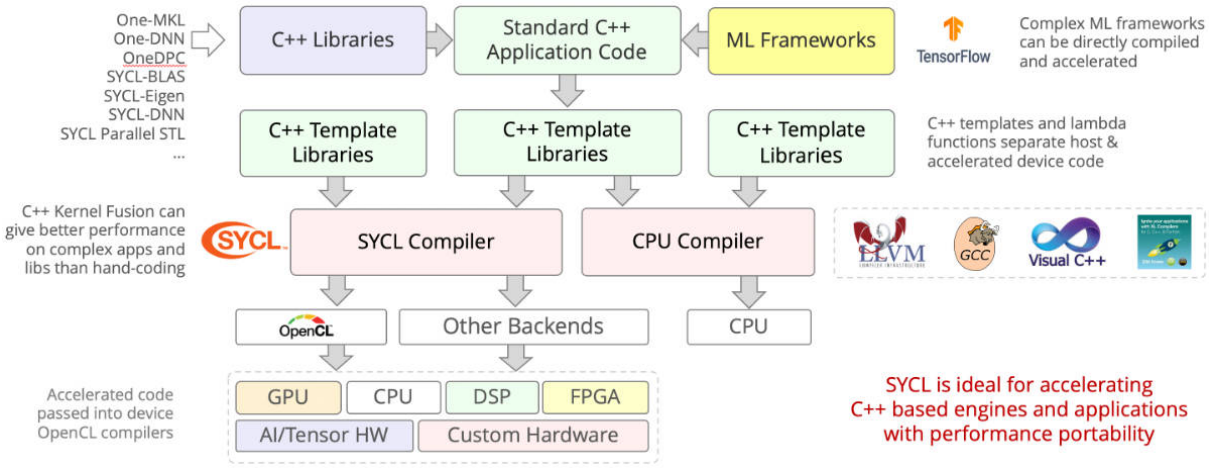

Figure 1: Ekosystem SYCL

Obsługa urządzeń

Pierwszym etapem wykorzystania standardu *SYCL* jest dostęp do urządzenia. Dostęp do urządzenia w *SYCL* obejmuje kilka kroków:

1. Utworzenie obiektu platformy: Pierwszym krokiem jest utworzenie obiektu platformy, który reprezentuje podstawową platformę sprzętową i programową, na której będzie działać aplikacja SYCL. Można to zrobić za pomocą klasy *sycl::platform*.
2. Zapytanie o dostępne urządzenia: Po utworzeniu obiektu platformy można zapytać go o dostępne urządzenia za pomocą metody *get_devices()*. Metoda ta zwraca wektor obiektów *sycl::device* reprezentujących wszystkie dostępne urządzenia na platformie.

3. Wybór urządzenia: Po wyszukaniu dostępnych urządzeń należy wybrać jedno lub więcej urządzeń, które będą używane do wykonywania jądra. Można to zrobić poprzez utworzenie instancji *sycl::device* i przekazanie jej do konstruktora kolejki lub poprzez podanie selektora urządzenia.
4. Utworzenie kolejki: Po wybraniu jednego lub więcej urządzeń należy utworzyć obiekt kolejki, który będzie zarządzał wykonywaniem kerneli na tych urządzeniach. Można to zrobić za pomocą klasy *sycl::queue*.

Fragment kodu przedstawiający dostęp do urządzania typu *GPU*:

```
    queue q(gpu_selector_v);
    cout << "[Device]\t" << q.get_device().get_info<info::device::name>()
↪   << "\n";
```

Kolejki

W *SYCL*, kolejka to obiekt zarządzający wykonaniem kerneli na urządzeniu. Kolejka enkapsuluje pojedyncze urządzenie i umożliwia przesyłanie grup poleceń do wykonania przez środowisko SYCL.

Tworząc obiekt kolejki, można określić, które urządzenie ma być zarządzane, używając instancji *sycl::device* lub *selektora urządzenia* (*device selector*). Po utworzeniu kolejki, zadania można dodawać do niej, tworząc jedną lub więcej instancji *sycl::kernel* i wywołując metodę *submit()* kolejki, podając te kernele jako argumenty.

Środowisko SYCL zaplanuje i wykona te kernele na wybranych urządzeniach w kolejności, w jakiej zostały przesłane. Kolejka udostępnia również metody do synchronizacji danych między pamięcią hosta a pamięcią urządzenia, oraz do pobierania informacji o podstawowych właściwościach urządzenia, takich jak nazwa, producent i możliwości.

Warto zauważyć, że kolejki działają asynchronicznie, co oznacza, że nie blokują się podczas wykonywania kerneli. Zadania są przekazywane do kolejki i wykonują się w tle, podczas gdy inne operacje są wykonywane na CPU hosta. Aby upewnić się, że wszystkie zadania w kolejce zostały wykonane przed zakończeniem programu, można użyć *eventów* lub barier do synchronizacji między pamięcią hosta a pamięcią urządzenia.

Kolejki są istotnym elementem programowania w SYCL, ponieważ umożliwiają zarządzanie wykonaniem kernel na urządzeniach. Poprzez efektywne wykorzystanie kolejek i zrozumienie ich zachowania i możliwości, można pisać wydajne i skalowalne aplikacje SYCL, w pełni wykorzystując nowoczesne architektury sprzętowe.

Podsumowanie metod dostępnych dla kolejek:

1. **submit()**: Metoda ta służy do przesłania grupy poleceń do wykonania przez runtime SYCL na urządzeniu zarządzanym przez kolejkę. Jako argumenty przyjmuje jedną lub więcej instancji klasy kernel, które reprezentują jądra, które mają być wykonane.
2. **wait()**: Metoda ta blokuje wywołujący wątek do czasu zakończenia wszystkich zadań znajdujących się w kolejce. Może być używana do synchronizacji między pamięcią hosta a urządzeniem.
3. **get_device()**: Metoda ta zwraca obiekt reprezentujący urządzenie zarządzane przez kolejkę.
4. **is_host()**: Metoda ta zwraca wartość logiczną określającą, czy kolejka jest kolejką hosta (czyli czy zadania są wykonywane na CPU) czy nie.
5. **throw_asynchronous()**: Metoda ta powoduje zgłoszenie wyjątku asynchronicznego, który może być przechwycony przez blok catch w innym wątku.
6. **get_info()**: Metoda ta zwraca informacje o kolejce, takie jak jej nazwa, dostępność funkcji specyficznych dla danego urządzenia, itp.
7. **submit_barrier()**: Metoda ta przesyła do kolejki polecenie oczekiwania na zakończenie wszystkich zadań znajdujących się w kolejce.
8. **submit_marker()**: Metoda ta przesyła do kolejki polecenie umieszczenia znacznika, który może być użyty do synchronizacji między hostem a urządzeniem.

Fragment kodu przedstawiający wykorzystanie kolejki do aceleracji obliczeń:

```
q.submit([&](auto &h)
{
    accessor a(a_buf, h, read_only);
    accessor b(b_buf, h, read_only);
    accessor c(c_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(range(m, p), [=](auto index)
    {
        int row = index[0];
        int col = index[1];
        auto sum = 0.0;
        for (int i = 0; i < n; i++)
            sum += a[row * n + i] * b[i * p + col];
        c[row * p + col] = sum;
    });
});
q.wait();
```

Pamięć

W *SYCL* dostęp do pamięci jest realizowany za pomocą tzw. *USM* (Unified Shared Memory), czyli jednolitej pamięci współdzielonej. *USM* umożliwia programistom zarządzanie pamięcią w sposób niezależny od urządzenia i systemu operacyjnego.

Do zarządzania pamięcią w *SYCL* wykorzystuje się dwa rodzaje obiektów: *USM pointers* i bufory.

USM pointers to wskaźniki, które pozwalają na bezpośredni dostęp do pamięci hosta lub urządzenia. W przypadku hosta, wskaźnik ten wskazuje na obszar pamięci RAM hosta, a w przypadku urządzenia - na obszar pamięci VRAM urządzenia. Dzięki temu programiści mogą bezpośrednio manipulować danymi znajdującymi się w pamięci hosta lub urządzenia.

Bufor to obiekt, który umożliwiaia programistom zarządzanie danymi znajdującymi się w pamięci hosta lub urządzenia. Bufory są tworzone za pomocą konstruktora *buffer()* i przyjmują jako argumenty rozmiar bufora oraz wskaźnik do danych. Bufory pozwalają na przesyłanie danych między hostem a urządzeniem oraz między różnymi urządzeniami.

Fragment kodu przedstawiający wykorzystanie obiektu *buffer* w celu dostępu do pamięci hosta:

```
buffer<float> a_buf(a, m * n);
buffer<float> b_buf(b, n * p);
buffer<float> c_buf(c, m * p);

q.submit([&](auto &h)
{
    accessor a(a_buf, h, read_only);
    accessor b(b_buf, h, read_only);
    accessor c(c_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(range(m, p), [=](auto index)
    {
        // a,b,c są poprawnymi wskaźnikami w tym domknięciu
    });
});
```

Co dalej?

Przedstawione wyżej funkcje standardu *SYCL* umożliwiają nam wygodne przyspieszanie różnorodnych obliczeń. Jednakże, ta forma akceleracji ma swoje ograniczenia. W pewnym momencie możemy napotkać granice wydajności naszych akceleratorów, co uniemożliwi dalsze skalowanie wertykalne. Użycie coraz bardziej zaawansowanych i szybszych akceleratorów nie zawsze jest możliwe i rzadko jest najlepszym rozwiązaniem.

Z tego powodu, złożone obliczenia muszą być skalowane horyzontalnie poprzez równoległe działanie wielu maszyn nad tym samym problemem. Takie podejście jednak wiąże się z pewnym problemem, którego wcześniej mogliśmy nie dostrzec - lokalnością danych.

Lokalność danych i obliczeń

W uproszczeniu, lokalność można opisać jako miarę opóźnienia, które doświadcza nasza jednostka obliczeniowa podczas pobierania kolejnych danych. W przypadku komputerów stacjonarnych, ten problem jest niezauważalny ze względu na niewielką liczbę poziomów pamięci. Ogólnie rzecz biorąc, mamy bardzo bliską pamięć podręczną (cache) oraz bardziej odległą pamięć operacyjną (RAM).

W przypadku skalowania horyzontalnego i wykorzystania akceleratorów, pojawia się wiele dodatkowych poziomów lokalności - pamięć może znajdować się na innym węźle obliczeniowym. Wykorzystanie interfejsów sieciowych wprowadza znaczące obciążenie w porównaniu do dostępu do pamięci RAM, co należy uwzględnić podczas tworzenia algorytmów dla takich systemów.

Skalowanie horyzontalne

Skalowanie horyzontalne jest techniką, która umożliwia zwiększanie mocy obliczeniowej poprzez równoległe uruchamianie aplikacji na wielu maszynach. Jednym z popularnych narzędzi wykorzystywanych do realizacji tego rodzaju skalowania jest biblioteka *MPI* (Message Passing Interface).

MPI to interfejs komunikacyjny, który umożliwia wymianę danych między procesami działającymi na różnych maszynach. Pozwala na podział zadania na mniejsze podzadania, które mogą być przetwarzane niezależnie na poszczególnych węzłach obliczeniowych, a następnie wymieniać między sobą niezbędne informacje.

Wykorzystując *MPI*, możemy skalować nasze obliczenia horyzontalnie poprzez uruchomienie równoległych instancji aplikacji na wielu maszynach w klastrze. Każda instancja może być odpowiedzialna za przetwarzanie fragmentu danych, a komunikacja między nimi odbywa się za pomocą funkcji komunikacyjnych udostępnianych przez *MPI*.

Jednym z kluczowych aspektów skalowania horyzontalnego przy użyciu *MPI* jest odpowiednie rozłożenie zadań i danych pomiędzy procesy. Właściwy podział pracy pozwala na równomierne wykorzystanie zasobów każdej maszyny i minimalizuje czas komunikacji między nimi.

Dzięki wykorzystaniu skalowania horyzontalnego z użyciem *MPI* możemy efektywnie wykorzystać potencjał wielu maszyn do przyspieszenia naszych obliczeń. Jednakże, konieczne jest świadome

projektowanie aplikacji, uwzględniające zarówno aspekty komunikacyjne, jak i odpowiednie rozłożenie obciążenia, aby osiągnąć optymalne wyniki.

Przykładowe funkcje udostępniane przez implementacje *MPI*, które zostały wykorzystane w kodzie prezentowanym na zajęciach:

1. **MPI_Init(NULL, NULL)**: Inicjalizuje środowisko MPI. Ta funkcja musi być wywołana jako pierwsza przed użyciem innych funkcji MPI.
2. **MPI_Get_processor_name(hostname, &len)**: Pobiera nazwę hosta, na którym wykonuje się dany proces MPI, i zapisuje ją w zmiennej `hostname`. Parametr `len` wskazuje na rozmiar bufora `hostname` i po wywołaniu funkcji zawiera rzeczywisty rozmiar nazwy hosta.
3. **MPI_Comm_size(MPI_COMM_WORLD, &world_size)**: Zwraca liczbę procesów w komunikatorze `MPI_COMM_WORLD` i zapisuje wynik do zmiennej `world_size`.
4. **MPI_Comm_rank(MPI_COMM_WORLD, &world_rank)**: Zwraca numer identyfikacyjny bieżącego procesu w komunikatorze `MPI_COMM_WORLD` i zapisuje wynik do zmiennej `world_rank`.
5. **MPI_Send(a, m * n, MPI_FLOAT, 1, 0, MPI_COMM_WORLD)**: Wysyła dane znajdujące się w buforze a o rozmiarze `m * n` jako tablicę elementów typu `MPI_FLOAT` do procesu o numerze identyfikacyjnym `1` w komunikatorze `MPI_COMM_WORLD` z etykietą `0`.
6. **MPI_Recv(a, m * n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)**: Odbiera dane typu `MPI_FLOAT` o rozmiarze `m * n` do bufora a od procesu o numerze identyfikacyjnym `0` w komunikatorze `MPI_COMM_WORLD` z etykietą `0`. Parametr `MPI_STATUS_IGNORE` oznacza, że informacje o statusie otrzymanego komunikatu są ignorowane.
7. **MPI_Finalize()**: Kończy działanie środowiska MPI. Ta funkcja powinna być wywołana jako ostatnia w programie, po zakończeniu wszystkich operacji MPI.

Uruchamianie obliczeń na klastrze

Kubernetes jest popularnym systemem orkiestracji kontenerów, który umożliwia zarządzanie i wdrażanie aplikacji w klastrach. Dzięki Kubernetes można skonfigurować klastry, które składają się z wielu węzłów obliczeniowych, a następnie uruchamiać i zarządzać aplikacjami w tych klastrach.

W kontekście obliczeń rozproszonych, można wykorzystać Kubernetesa do uruchomienia wielu instancji aplikacji wykonujących równoległe obliczenia na różnych węzłach klastra. Można utworzyć replikowane podstawowe zadania obliczeniowe, które będą pracować niezależnie na różnych węzłach. Kubernetes zajmuje się zarządzaniem skalowaniem, przydzielaniem zasobów i automatycznym przywracaniem w przypadku awarii.

Wykorzystanie Kubernetesa do uruchamiania takich obliczeń w rozproszonym środowisku pozwala na łatwą skalowalność, zarządzanie i monitorowanie aplikacji. Można dynamicznie dostosowywać liczbę instancji aplikacji w zależności od obciążenia i dostępnych zasobów. Ponadto, Kubernetes oferuje mechanizmy do komunikacji między instancjami aplikacji, takie jak usługi sieciowe, które umożliwiają wymianę danych między różnymi częściami aplikacji działającymi na różnych węzłach klastra.

Prezentowany przykład wykorzystywał dwa węzły obliczeniowe do wykonania obliczeń na GPU. Aby uruchomić takie obliczenia musimy opisać odpowiednio nasze zadanie obliczeniowe i przekazać do klastra. Możemy to zrobić za pomocą pliku *yaml*:

```yaml
apiVersion: kubeflow.org/v2beta1
kind: MPIJob
metadata:
  name: example
spec:
  slotsPerWorker: 1
  runPolicy:
    cleanPodPolicy: Running
  sshAuthMountPath: /home/mpiuser/.ssh
  mpiImplementation: Intel
  mpiReplicaSpecs:
    Launcher:
      replicas: 1
      template:
        spec:
          containers:
          - image: ghcr.io/blazej-smorawski/example-sycl-mpi:latest
            imagePullPolicy: Always
            name: mpi-launcher
            securityContext:
              runAsUser: 1000
              runAsGroup: 109
            args:
            - mpirun
            - -genv
            - I_MPI_DEBUG=2
            - -n
            - "2"
            - /home/mpiuser/example
            - "2000"
            - "2000"
            - "2000"
            resources:
```

```
            limits:
              cpu: 1
              memory: 4Gi
  Worker:
    replicas: 2
    template:
      spec:
        containers:
        - image: ghcr.io/blazej-smorawski/example-sycl-mpi:latest
          imagePullPolicy: Always
          name: mpi-worker
          securityContext:
            runAsUser: 1000
            runAsGroup: 109
          command:
          args:
          - /usr/sbin/sshd
          - -De
          - -f
          - /home/mpiuser/.sshd_config
          resources:
            limits:
              cpu: 1
              memory: 4Gi
              gpu.intel.com/i915: 1
```

Warto zwrócić uwagę na kilka elementów tego pliku:

1. Uruchamiane polecenie to `mpirun -genv I_MPI_DEBUG=2 -n 2 /home/mpiuser/example 2000 2000 2000`, co oznacza, że nasze zadanie zostanie wykonane na dwóch węzłach obliczeniowych z tymi samymi parametrami. `I_MPI_DEBUG=2` dostarcza nam nieco dodatkowych informacji o tym, jak przebiega komunikacja wykorzystująca interfejs *MPI*.
2. Wykorzystujemy zasób `gpu.intel.com/i915: 1`, aby mieć pewność że każdy uruchomiony kontener będzie miał dostęp do akceleratora gpu.
3. Nasze kontenery są uruchamiane jako `runAsGroup: 109`, ponieważ jest to numer grupy która ma dostęp do GPU na węzłach tego klastra.

Wyniki pracy klastra *k8s* nad tym zadaniem:

```
[0] MPI startup(): Intel(R) MPI Library, Version 2021.9  Build 20230307
↪  (id: d82b3071db)
[0] MPI startup(): Copyright (C) 2003-2023 Intel Corporation.  All rights
↪  reserved.
```

```
[0] MPI startup(): library kind: release
[0] MPI startup(): libfabric version: 1.13.2rc1-impi
[0] MPI startup(): libfabric provider: tcp;ofi_rxm
[0] MPI startup(): File "/opt/intel/oneapi/mpi/2021.9.0/etc/tuning_skx_shm-
↪  ofi_tcp-ofi-rxm_10.dat" not
↪  found
[0] MPI startup(): Load tuning file:
↪  "/opt/intel/oneapi/mpi/2021.9.0/etc/tuning_skx_shm-ofi_tcp-ofi-rxm.dat"
[HOST]      example-worker-1
[HOST]      example-worker-0
[Device]    Intel(R) Iris(R) Graphics 540 [0x1926]
[Device]    Intel(R) Iris(R) Graphics 540 [0x1926]
[FINAL]
[MPI]       Elapsed(ms)=829
[RESULT]    CORRECT
```

Dodatek: Korzystanie z oneAPI (SYCL + MPI + level-zero)

Aby skompilować i uruchomić powyżej opisane fragmenty kodu, możemy wykorzystać pakiet *oneAPI* dostarczany przez firmę *Intel*. Najwygodniejszym sposobem korzystania z tego pakietu oprogramowania jest użycie gotowych obrazów systemu.

Przykładowy obraz wykorzystany podczas prezentacji:

```
FROM intel/oneapi-basekit

WORKDIR /example

COPY example.cpp example.cpp

RUN icpx -fsycl example.cpp -O3 -o example

ENTRYPOINT ["./example"]
```

Dodatek: Instalacja implementacji MPI na systemach z rodziny *Debian*

Instalacja środowiska *MPI* i sterowników do kart graficznych firmy **Intel** na systemach operacyjnych korzystających z menadżera pakietów *apt* jest bardzo prosta i wymaga kilku prostych poleceń:

Pierwszym etapem jest instalacja repozytorium **Intel(R) oneAPI**

```
apt-get update && apt-get upgrade -y && \
DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
curl ca-certificates gnupg gpg-agent software-properties-common && \
rm -rf /var/lib/apt/lists/*

curl -fsSL https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-
↪  PRODUCTS-2023.PUB| apt-key add -
↪  \
echo "deb [trusted=yes] https://apt.repos.intel.com/oneapi all main " >
↪  /etc/apt/sources.list.d/oneAPI.list
```

Następnie możemy zainstalować repozytorium **Intel(R) GPU** jeśli planujemy wykorzystywać karty graficzne firmy **Intel**:

```
apt-get update && apt-get upgrade -y && \
DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
curl ca-certificates gnupg gpg-agent software-properties-common && \
rm -rf /var/lib/apt/lists/*

curl -fsSL https://repositories.intel.com/graphics/intel-graphics.key |
↪  apt-key add -
echo "deb [trusted=yes arch=amd64]
↪  https://repositories.intel.com/graphics/ubuntu focal-devel main" >
↪  /etc/apt/sources.list.d/intel-graphics.list
```

Po zainstalowaniu odpowiednich repozytoriów możemy zainstalować wybrane przez nas pakiety. W tym przykładzie wybraliśmy paczkę *runtime*, która zawiera środowisko uruchomieniowe dla *oneAPI*, czyli między innymi biblioteki oraz pliki wykonywalne potrzebne dla działania *MPI*. Dodatkowo instalujemy *level-zero*, czyli bardzo przystępny sterownik pozwalający na wykorzystywanie akceleracji *GPU*.

```
apt-get update && apt-get upgrade -y && \
DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
sudo intel-oneapi-runtime-dpcpp-cpp intel-level-zero-gpu level-zero && \
rm -rf /var/lib/apt/lists/*
```

Źródła

Kod źródłowy wykorzystany do powyżej opisanych przykładów oraz gotowe kontenry znajdują się w repozytorium *git*: https://github.com/blazej-smorawski/oneapi-hackathon

Dodatek: Uruchamianie *MPI* w środowisku *k8s*

Kroki potrzebne do uruchomienia *MPI* w środowisku *kubernetes*:

1. Instalacja *MPI-operator*

```
kubectl apply -f https://raw.githubusercontent.com/kubeflow/mpi-
↪  operator/v0.4.0/deploy/v2beta1/mpi-operator.yaml
```

2. Przygotowanie odpowiedniego obrazu zawierającego odpowienie środowisko *ssh*, *MPI* oraz nasz plik wykonywalny:

```
ARG BASE_LABEL

FROM mpioperator/intel-builder:${BASE_LABEL} as builder

COPY example-mpi.cpp /src/example-mpi.cpp
RUN bash -c "source /opt/intel/oneapi/setvars.sh && icpx -fsycl -lmpi
↪  /src/example-mpi.cpp -O3 -o /example"

FROM mpioperator/intel:${BASE_LABEL}

# Instalacja *MPI* oraz *level-zero* w ten sam sposób jak powyżej

COPY --from=builder /example /home/mpiuser/example
```

3. Budowanie obrazu:

```
docker build . -t ghcr.io/blazej-smorawski/example-sycl-cpu
```

4. Publikacja obrazu, aby był on dostępny dla każdego węzła obliczeniowego:

```
docker push ghcr.io/blazej-smorawski/example-sycl-cpu
```

5. Stworzenie pliku zawierającego definicję *custom-resource MPIJob*
6. Uruchomienie obliczeń:

```
kubectl apply -f k8s-job.yaml
```

7. Pracę naszego zadania możemy obserwować za pomocą:

```
kubectl describe mpijobs.kubeflow.org
```