Politechnika Świętokrzyska
w Kielcach
Wydział Elektrotechniki, Automatyki i Informatyki

Współczesne architektury procesorów graficznych

Instrukcja laboratoryjna 1

„Wprowadzenie do GPGPU Podstawy OpenCL”

Przygotował:
mgr inż. Paweł Pięta

Kielce, 2019

1 GPGPU

Przyspieszenie pracy programu komputerowego i skrócenie czasu jego wykonania może nastąpić na trzy główne sposoby:

1. zastosowanie procesora o szybszym taktowaniu,
2. zastosowanie procesora wykonującego większą liczbę instrukcji na jeden cykl zegara,
3. zastosowanie większej liczby procesorów.

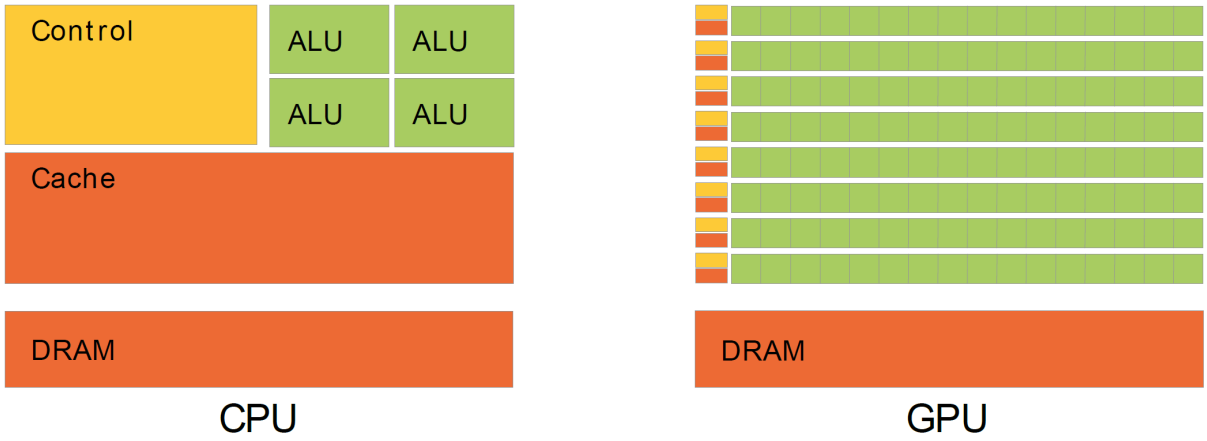
W przypadku punktów 1 i 2 został osiągnięty pewien limit rozwoju sprzętu, przez co nie ma możliwości dalszego, łatwego skalowania wydajności w tych kierunkach. Dlatego najlepszym wyjściem dla programisty jest odwołanie się do obliczeń równoległych. Jednym z dostępnych rozwiązań jest wykorzystanie procesorów graficznych.

Pierwsze GPU (ang. *Graphics Processing Unit*) zostały zaprojektowane jako akceleratory graficzne, wspierające jedynie stały graficzny potok przetwarzania danych (ang. *fixed graphics pipeline*). Pojęcie potoku renderującego pojawiło się na początku 1999 roku wraz z premierą API DirectX 6.1. Pod koniec tego samego roku karty graficzne zostały wyposażone w układ T&L (inaczej nazywany TCL). Był on odpowiedzialny za sprzętowe przekształcanie współrzędnych renderowanych obiektów (*transform*), pozostawianie jedynie obiektów widocznych dla obserwatora (*clipping*), oraz wykonywanie obliczeń związanych z oświetleniem obiektów (*lighting*). Dostęp do T&L realizowany był poprzez API DirectX 7.0.

W 2000 roku, za sprawą wprowadzonych w API DirectX 8.0 shaderów, które zastąpiły technologię T&L, karty graficzne stały się programowalne. Model przetwarzania grafiki oparty o shadery umożliwia programiście przeprowadzanie operacji na wierzchołkach sceny, a także na poszczególnych pikselach obrazu wynikowego, za pomocą specjalnych fragmentów kodu, wykonywanych w całości przez dedykowane jednostki strumieniowe (ang. *SPUs, Stream Processing Units*) wbudowane w akcelerator graficzny (klasa architektury SIMD[1]). Naukowcy zaczęli wykorzystywać moc GPU do obliczeń na liczbach zmiennoprzecinkowych – w ten sposób narodził się nurt GPGPU (ang. *General-Purpose Computing on Graphics Processing Units*), czyli wykonywanie obliczeń ogólnego przeznaczenia z użyciem procesorów graficznych. Jednakże podstawową niedogodnością tamtych czasów była konieczność mapowania obliczeń naukowych na problemy, które można wyrazić za pomocą trójkątów i wielokątów (np. za pomocą API OpenGL). Dlatego też w 2003 roku grupa naukowców na Uniwersytecie Stanforda pod kierownictwem Iana Bucka opracowała BrookGPU – pierwszy szeroko przyjęty model programowania, rozszerzający język C za pomocą specjalnych konstrukcji do równoległego przetwarzania danych. Używając koncepcji takich jak strumienie (*streams*), kernele (*kernels*) i operatory redukcji (*reduction operators*), BrookGPU za pomocą wysokopoziomowego API zapewnił dostęp do karty

[1] ang. *Single Instruction Multiple Data*

graficznej jako strumieniowego procesora ogólnego zastosowania. Programy korzystające z GPU stały się nie tylko łatwiejsze do napisania, ale także działały dużo szybciej w porównaniu z podobnym dostępnym wówczas kodem.



Rysunek 1: Schematyczne porównanie liczby tranzystorów w CPU i GPU[a]

[a]Źródło: CUDA C Programming Guide

Napędzane nienasyconym zapotrzebowaniem rynku na wysokiej jakości grafikę 3D, programowalne procesory graficzne ewoluowały w wysoce równoległe, wielowątkowe, wielordzeniowe procesory z ogromną mocą obliczeniową i bardzo dużą przepustowością pamięci. Są projektowane w taki sposób, aby więcej tranzystorów było w nich poświęconych przetwarzaniu danych, aniżeli ich buforowaniu, czy kontroli sterowania, co schematycznie pokazuje rysunek 1.

2 Paradygmat programowania GPGPU

Podczas zajęć do programowania procesorów graficznych będą wykorzystywane dwa języki programowania: OpenCL i CUDA. U rdzenia równoległego modelu programowania stoją trzy kluczowe abstrakcje:

- hierarchia wątków,
- hierarchia pamięci,
- synchronizacja wątków.

Wyeksponowane są one programiście w prosty sposób, jako minimalny zestaw rozszerzeń języka programowania. Głównym zadaniem programisty jest podział problemu na gruboziarniste podproblemy, które mogą zostać rozwiązane równolegle i niezależnie od siebie przez grupy wątków. Z kolei każdy podproblem powinien zostać podzielony na mniejsze, drobnoziarniste, których rozwiązaniem zajmą się wspólnie wątki jednej grupy, uruchomione równolegle. Taka forma dekompozycji zachowuje ekspresyjność języka, pozwalając

wątkom na współpracę podczas rozwiązywania każdego z podproblemów, jednocześnie umożliwiając automatyczną skalowalność.

Zarówno OpenCL jak i CUDA pozwalają programiście na definiowanie funkcji nazywanych kernelami – wywołane, wykonują się równolegle n-razy w n różnych wątkach. Liczba wątków, która wykona dany kernel, precyzowana jest w momencie jego wywołania. Każdy wątek otrzymuje unikalny identyfikator, którego pobranie możliwe jest wewnątrz kernela.

Model programowania GPGPU zakłada, że wątki wykonywane są na fizycznie oddzielnym urządzeniu (ang. *device*), które działa dla gospodarza (ang. *host*) na zasadzie koprocesora. Kod szeregowy wykonywany jest przez gospodarza, natomiast intensywne obliczeniowo fragmenty zrównoleglane są na urządzeniu. Zarówno gospodarz, jak i urządzenie, posiadają swoją własną przestrzeń pamięci, oznaczaną odpowiednio jako pamięć gospodarza i pamięć urządzenia. Możliwa jest alokacja/dealokacja obszarów pamięci urządzenia, jak również kopiowanie danych pomiędzy pamięcią gospodarza a pamięcią urządzenia. Zarówno hierarchia wątków, jak również hierarchia pamięci oraz synchronizacja wątków, zostaną dokładniej omówione w kolejnych instrukcjach.

Ogólny schemat działania każdego programu wykonującego obliczenia równoległe z zastosowaniem procesora graficznego wygląda następująco:

1. alokacja pamięci dla buforów danych po stronie gospodarza i urządzenia,
2. skopiowanie danych do obliczeń z pamięci gospodarza do pamięci urządzenia,
3. wywołanie kernela na urządzeniu w celu wykonania obliczeń równoległych,
4. skopiowanie rezultatów z pamięci urządzenia do pamięci gospodarza,
5. przeanalizowanie rezultatów przez gospodarza.

3 Podstawy OpenCL

OpenCL (*Open Computing Language*) to otwarty, darmowy, multiplatformowy standard programowania obliczeń równoległych, którego specyfikacja zarządzana jest i rozwijana przez konsorcjum Khronos Group[2].

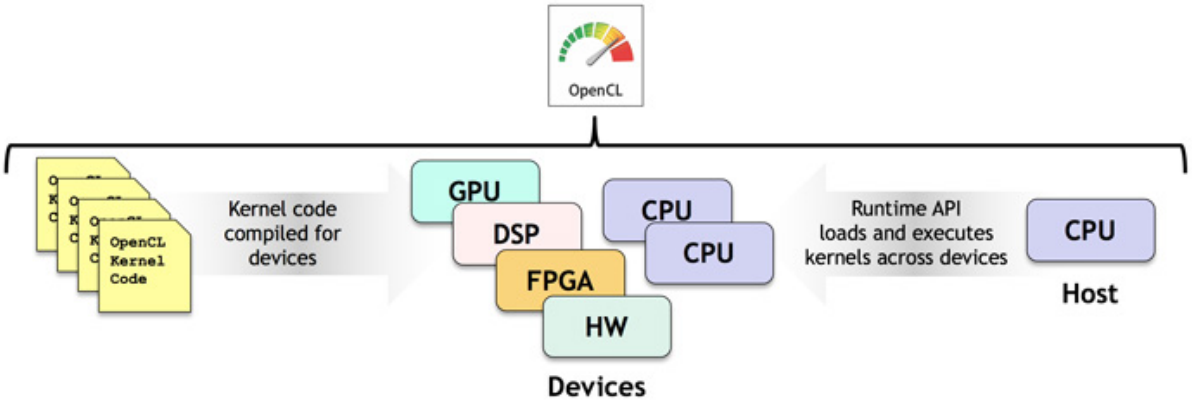
3.1 Wprowadzenie

OpenCL wspierany jest przez różnorodne procesory dostępne w komputerach osobistych (CPU, GPU), serwerach, urządzeniach mobilnych i systemach wbudowanych. Do uruchamiania kodu języka OpenCL nie służą jedynie procesory graficzne, ale również procesory sygnałowe (ang. *DSPs, Digital Signal Processors*), czy układy FPGA (patrz rysunek 2). W sercu języka znajdują się trzy API oraz język programowania kerneli:

- C Platform Layer API – pobieranie dostępnych urządzeń, wybieranie ich i inicjacja,

[2]Konsorcjum Khronos Group jest także odpowiedzialne za specyfikacje takich języków jak Vulkan, OpenGL, czy WebGL.

- C Runtime API – budowanie i uruchamianie kerneli na wielu urządzeniach,
- OpenCL C++ Wrapper API – opakowanie dwóch powyższych API w języku C++,
- OpenCL C – język pisania równoległego kodu kerneli oparty o język C.



Rysunek 2: API OpenCL i wspierane urządzenia[a]

[a]Źródło: `https://www.khronos.org/assets/uploads/apis/2015-api-opencl-1.png`

Na stronie internetowej `https://www.khronos.org/opencl/` opublikowana jest pełna dokumentacja języka, którego najnowsza wersja oznaczona jest numerem 2.2. W 2016 roku, wraz z wprowadzeniem wersji 2.2, dodano wsparcia dla programowania kerneli w języku C++14. Podczas zajęć OpenCL będzie wykorzystywany w standardzie nie wyższym niż 1.1.

3.2 Instalacja i konfiguracja

W celu skorzystania z języka OpenCL należy zainstalować:

- w przypadku kart graficznych firmy NVIDIA:
  - sterownik graficzny ze strony `https://www.geforce.com/drivers`,
  - CUDA Toolkit ze strony `https://developer.nvidia.com/cuda-zone`,
- w przypadku kart graficznych firmy AMD:
  - sterownik graficzny ze strony `https://www.amd.com/en/support`,
  - AMD OpenCL APP SDK lub ROCm (Radeon Open Compute Platform); pierwsze oprogramowanie nie jest już udostępniane przez AMD, natomiast drugie można pobrać ze strony `https://rocm.github.io`) – niestety jest ono wspierane dopiero od architektury Fiji.

Następnie w przypadku systemu operacyjnego Windows w IDE Microsoft Visual Studio należy utworzyć nowy, pusty projekt aplikacji konsolowej w języku C++, a w jego właściwościach odpowiednio skonfigurować kompilator i linker. Dla kart NVIDIA zostało to przedstawione na rysunkach 3, 4 i 5. Konfiguracja ta oparta jest o zmienną środowiskową

CUDA_PATH, tworzoną przez CUDA Toolkit w chwili jego instalacji. Ostatnim krokiem jest włączenie w kodzie programu dyrektywą #include biblioteki <CL/cl.h>.

3.3 Kernel

Funkcja kernela w języku OpenCL deklarowana jest za pomocą kwalifikatora deklaracji __kernel. Prosty przykład kernela wykonującego zerowanie wektora przedstawia kod źródłowy 1. Przez parametry przyjmuje on wskaźnik na bufor danych zaalokowany w pamięci urządzenia (oznaczony kwalifikatorem __global) oraz liczbę elementów wektora. Każdy wątek, który wykonuje kernel zeroVec(), przeprowadza zerowanie jednego elementu wektora pod indeksem równym swojemu unikalnemu identyfikatorowi, który wewnątrz kernela pobierany jest za pomocą funkcji get_global_id(). Kod kernela w języku OpenCL zazwyczaj przechowywany jest w osobnym pliku o rozszerzeniu cl, który wczytywany jest przez gospodarza w trakcie działania programu.

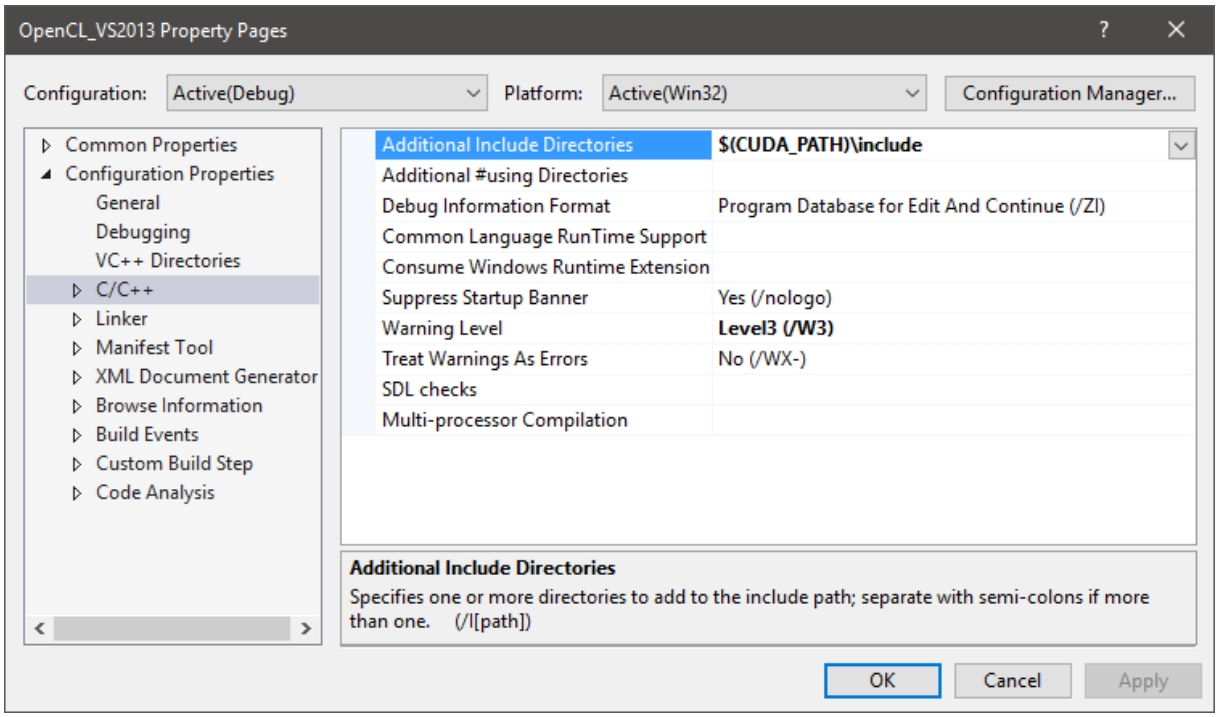
Kod źródłowy 1: Definicja kernela na przykładzie zerowania wektora

```
1  __kernel void zeroVec(__global int *vec, const uint size)
2  {
3      /* Get a global ID of the thread. */
4      uint id = get_global_id(0);
5
6      /* If the thread ID is within the range of the vector size,
7         zero a single element of the vector. */
8      if (id < size)
9          vec[id] = 0;
10 }
```

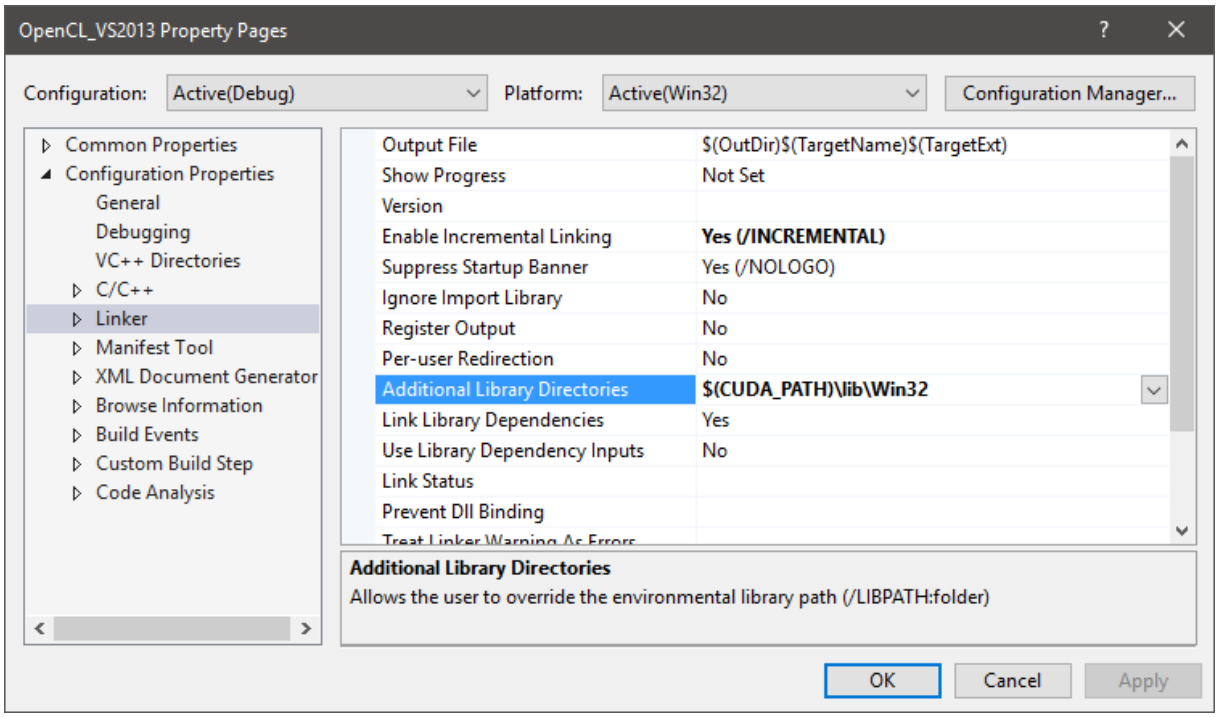
3.4 Struktura programu

Przed uruchomieniem kernela napisanego w OpenCL, konieczne jest wykonanie kilku rzeczy. W pierwszej kolejności za pomocą funkcji clGetPlatformIDs() należy pobrać listę dostępnych platform obliczeniowych. Następnie z użyciem funkcji clGetDeviceIDs() dla znalezionych numerów platform trzeba pozyskać listę dostępnych w nich urządzeń wspierających OpenCL. Ponieważ celem przedmiotu jest programowanie procesorów graficznych, jako drugi parametr device_type należy podać flagę CL_DEVICE_TYPE_GPU, co ograniczy rezultaty jedynie do kart graficznych.

Kolejnym krokiem jest wywołanie funkcji clCreateContext(), która dla wybranego urządzenia utworzy kontekst OpenCL. Służy on zarządzaniu takimi obiektami jak kolejki poleceń, zmienne dynamiczne przechowywane w pamięci urządzenia, obiekty programów i kerneli. Z jego użyciem są również uruchamiane kernele na urządzeniu. Aby dokonać tego ostatniego, za pomocą funkcji clCreateCommandQueue() należy najpierw utworzyć kolejkę poleceń dla wskazanego urządzenia.

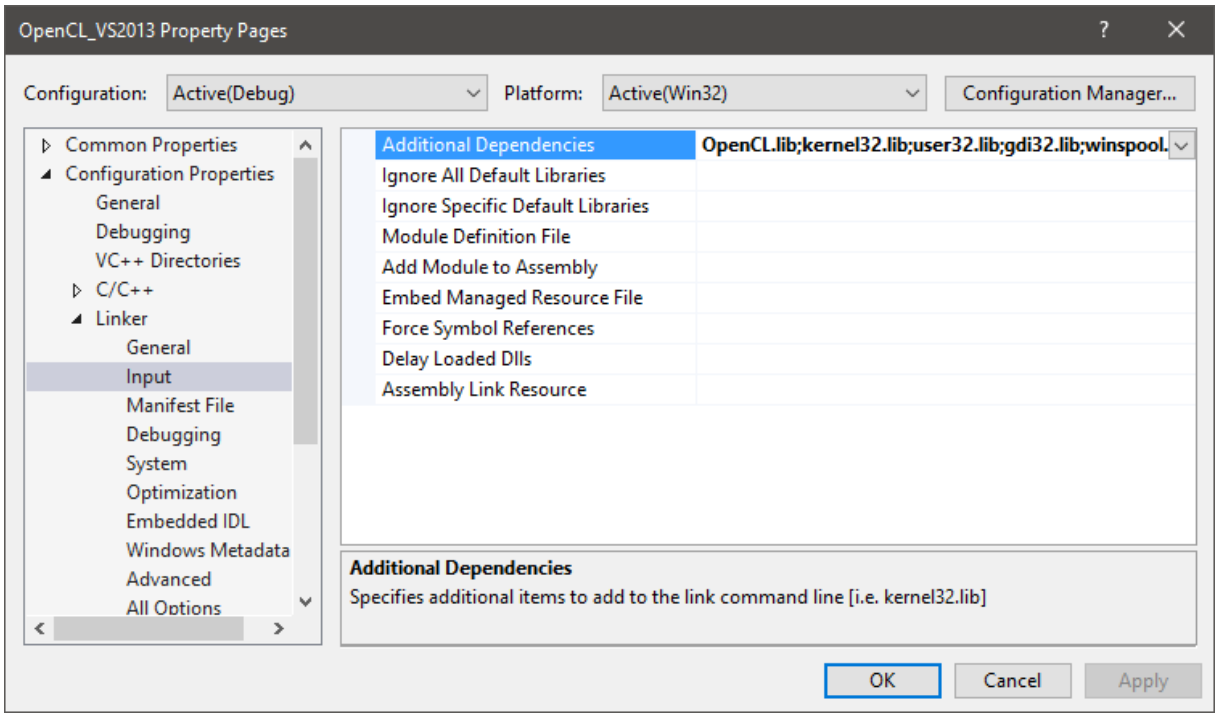


Rysunek 3: Konfiguracja ścieżki przeszukiwań kompilatora w ustawieniach projektu VS



Rysunek 4: Konfiguracja ścieżki przeszukiwań linkera w ustawieniach projektu VS

Posiadając kontekst i kolejkę poleceń, można przejść do przygotowania kernela. Najpierw należy wczytać z dysku kod źródłowy kernela i na jego podstawie z użyciem funkcji `clCreateProgramWithSource()` utworzyć obiekt programu, który następnie trzeba zbudować funkcją `clBuildProgram()`. Ostatnim krokiem jest utworzenie obiektu kernela poprzez wywołanie funkcji `clCreateKernel()`. Istotną rzeczą jest, aby za drugi parametr



Rysunek 5: Konfiguracja dodatkowych zależności linkera w ustawieniach projektu VS

`kernel_name` podać faktyczną nazwę funkcji kernela z kodu źródłowego.

Przed uruchomieniem kernela należy jeszcze zaalokować pamięć na potrzebne bufory danych po stronie urządzenia. Służy do tego funkcja `clCreateBuffer()`. W trakcie tych zajęć za parametr `flags` można podstawić flagę `CL_MEM_READ_WRITE`. Ostatnią rzeczą jest ustawienie argumentów wywołania dla obiektu kernela za pomocą funkcji `clSetKernelArg()`. Samo wywołanie kernela na urządzeniu realizowane jest z użyciem funkcji `clEnqueueNDRangeKernel()`. O liczbie wątków wewnątrz grupy wątków i łącznej liczbie wątków decydują parametry `local_work_size` i `global_work_size` – na potrzeby tej instrukcji pierwszy z nich przyjmuje wartość `NULL`, natomiast drugi ma wartość równą liczbie elementów przetwarzanego wektora. Utworzenie bufora, ustawienie argumentów i wywołanie kernela zostały pokazane na przykładzie kernela `zeroVec()` w kodzie źródłowym 2. Za pomocą funkcji `clFinish()` następuje wstrzymanie wykonania programu do momentu zakończenia przez urządzenie wszystkich operacji.

Przed zakończeniem pracy programu należy pozwalniać przydzieloną pamięć i wszystkie obiekty funkcjami z rodziny `clRelease*()`:

- `clReleaseMemObject()`,
- `clReleaseKernel()`,
- `clReleaseProgram()`,
- `clReleaseCommandQueue()`,
- `clReleaseContext()`.

Kod źródłowy 2: Utworzenie bufora, ustawienie argumentów i wywołanie kernela

```
cl_int err = 0;
cl_uint size = 100;
size_t bytes = size*sizeof(int);
size_t global_work_size = size;

cl_mem d_vec = clCreateBuffer(context, CL_MEM_READ_WRITE,
                              bytes, NULL, &err);
// Check for errors.
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_vec);
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint), &size);
// Check for errors.
err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                             &global_work_size, NULL, 0, NULL, NULL);
err = clFinish(command_queue);
// Check for errors.
```

4 Zadania

1. Pobierz, skompiluj i uruchom przykładowy program `OpenCL_VS2013`. Na jego podstawie zrealizuj kolejne zadania.

2. Napisz program w OpenCL, w którym w pamięci karty graficznej zostanie zaalokowany wektor elementów typu `int` o takiej długości, aby zająć prawie całą pojemność pamięci karty. Przykładowo, dla karty graficznej wyposażonej w 512 MB pamięci może być to wektor 125 milionów elementów (około 477 MB). Następnie uruchom 100 razy kernel zapisujący w tym wektorze wartość 1000 i zmierz czas wykonania programu, np. za pomocą funkcji `clock()` (pamiętaj o podzieleniu wyniku przez stałą `CLOCKS_PER_SEC`).

3. Powtórz powyższe zadanie dla wektora zaalokowanego w pamięci głównej, wypełnianego szeregowym kodem przez CPU. Porównaj czas wykonania obu programów. Uwaga! Konieczne będzie wykorzystanie dynamicznej alokacji pamięci.