Programowanie kart graficznych

Architektura i API
część 3

CUDA
hierarchia pamięci c.d.

Pamięć „page-locked”:

- środowisko udostępnia specjalizowane usługi do przydzielania i zwalniania pamięci typu „page-locked”
- bloki pamięci „page-locked” przydziela się w pamięci hosta
- pamięć taka nie podlega wymianie w trakcie normalnej pracy pamięci wirtualnej systemu hosta
- pamięć taka jest traktowana w szczególny sposób, umożliwiający prowadzanie transferów danych do/z niej równolegle z wykonaniem kodu urządzenia (dla niektórych urządzeń CUDA)
- pamięć „page-locked” może być mapowana w przestrzeń adresową urządzenia (dla niektórych urządzeń CUDA)

Pamięć „page-locked”:

- pamięć „page-locked” jest w każdym systemie zasobem bardzo deficytowym – spodziewaj się, że przydziały tej pamięci przestaną się udawać na długo przed wyczerpaniem możliwości przydziału „zwykłej” pamięci
- pamiętaj także, że przydzielenie pamięci „page-locked” w każdym przypadku obniża wydajność całego systemu
- blok pamięci „page-locked” może być używana w każdym z wątków, ale pełną korzyść z podwyższenia wydajności uzyskają tylko te wątki, które jawnie zapewnią sobie dostęp do takiej pamięci (parametr `cudaHostAllocPortable` w wywołaniu funkcji `cudaHostAlloc()`)
- jest to tzw. pamięć przenośna (ang. portable)

Pamięć „page-locked”:

- domyślnie pamięć „page-locked” podlega cache'owaniu przez pamięci podręczne procesora
- jeżeli zależy ci na tym, aby uwolnić cache na rzecz innych czasochłonnych czynności angażujących CPU (a przemieszczanie danych w pamięci do nich nie należy), możesz wyłączyć ten obszar ze sfery zainteresowania pamięci podręcznej
- uczynisz to przekazując parametr `cudaHostAllocWriteCombined` do funkcji `cudaHostAlloc()`
- wyłączenie działanie cache'a wg firmowej dokumentacji Nvidii przyspiesza transfer z urządzenia nawet do 40%...
- ... ale transfer do urządzenia może ulec spowolnieniu

Pamięć „page-locked”:

- wskazówka ogólna – używaj opcji `cudaHostAllocWriteCombined` w odniesieniu do pamięci, którą host intensywnie zapisuje, ale nie do pamięci, którą host intensywnie odczytuje

Pamięć mapowana :

- **niektóre** urządzenia CUDA umożliwiają przemapowanie części pamięci hosta do przestrzeni adresowej urządzenia
- taki blok pamięci identyfikowany jest przez dwa różne adresy:
  - jeden w przestrzeni adresowej hosta
  - drugi w przestrzeni adresowej urządzenia CUDA
- host uzyskuje swój adres wywołując funkcję `cudaHostAlloc()` z parametrem `cudaHostAllocMapped`
- urządzenie uzyskuje swój adres wywołując funkcję `cudaHostGetDevicePointer()`
- adres przypisany do użycia przez urządzenie może być następnie wykorzystany w treści kernela

Pamięć mapowana :

- pamięć mapowana pozwala uniknąć konieczności jawnego transferu danych pomiędzy hostem a urządzeniem
- transfer taki wykonywany jest niejawnie przez sprzęt urządzenia dopiero w chwili żądania dostępu aktywowanego w treści wątku urządzenia
- ze względu na możliwość równoczesnego wykonania kodu kernela i kodu hosta korzystających z tego samego bloku pamięci mapowanej, programista musi rozważyć wprowadzenie dodatkowych mechanizmów synchronizacyjnych (o tym niebawem)

Pamięć mapowana :

- blok pamięci „page-locked” może być jednocześnie mapowany i przenośny
- w takim przypadku każdy z wątków hosta (!) musi na swoją rzecz wywołać funkcję cudaHostGetDevicePointer() w celu uzyskania własnej kopii adresu tego bloku przeznaczonej do użycia przez urządzenie
- dzieje się tak dlatego, że wątki urządzenia uruchamiane z różnych wątków hosta używają (dokładniej - mogą używać) różnych adresów tego samego bloku pamięci mapowanej

Pamięć mapowana :

- aby umożliwić pobranie unikalnego adresu bloku pamięci mapowanej wewnątrz kodu wątku hosta, należy wykonać funkcję `cudaSetDeviceFlags()` z parametrem `cudaDeviceMapHost` zanim zostanie odpalony jakikolwiek wątek urządzenia
- w przeciwnym przypadku funkcja `cudaHostGetDevicePointer()` wywołana w wątku urządzenia zwróci błąd
- aplikacja może sprawdzić dostępność pamięci mapowanej w konkretnym egzemplarzy urządzenia posługując się funkcją `cudaGetDeviceProperties()` i sprawdzając wartość pola `canMapHostMemory`

Pamięć mapowana – API: przydział pamięci hosta

```
cudaError_t cudaMallocHost  (
   void           **ptr,
   size_t           size
)
```

Parametry:

- `ptr` - wskaźnik na wskaźnik do przydzielonego bloku (*wartość ustawiana przez funkcję*)
- `size` - żądany rozmiar bloku (bajty)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorMemoryAllocation` - błąd przydziału

Uwaga – funkcja przydziela się pamięć typu „page-locked”

Pamięć mapowana – API: przydział pamięci hosta

```
cudaError_t cudaHostAlloc(
   void          **ptr,
   size_t          size,
   unsigned int flags
)
```

Parametry:

- `ptr` - wskaźnik na wskaźnik do przydzielonego bloku (*wartość ustawiana przez funkcję*)
- `size` - żądany rozmiar bloku (bajty)
- `flags` - żądane własności przydzielanego bloku (maska)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorMemoryAllocation` - błąd przydziału

Pamięć mapowana – API: przydział pamięci hosta

opcje przydziału:

- **`cudaHostAllocDefault`**
  opcje domyślne – funkcja będzie zachowywać się dokładnie tak samo, jak cudaMallocHost
- **`cudaHostAllocPortable`**
  pamięć będzie traktowana jako przenośna (dostępna dla wszystkich wątków urządzenia)
- **`cudaHostAllocMapped`**
  pamięć będzie mogła być zamapowana przez wątki urządzenia
- **`cudaHostAllocWriteCombined`**
  pamięć wyłączona spod działania cache'a CPU

Pamięć mapowana – API: mapowanie pamięci hosta

```
cudaError_t cudaHostGetDevicePointer(
   void          ** pDevice,
   void           * pHost,
   unsigned int   flags
)
```

Parametry:

- `pDevice` - wskaźnik na wskaźnik zamapowany blok pamięci (*wartość ustawiana przez funkcję*)
- `pHost` - istniejący wskaźnik na blok pamięci hosta (uzyskany z `cudaHostAlloc()` lub `cudaMallocHost()`)
- `flags` - żądane własności mapowanego bloku (maska)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorMemoryAllocation` - błąd przydziału

Pamięć mapowana – API: mapowanie pamięci hosta

opcje przydziału:

- w chwili obecnej tylko i wyłącznie 0

Obsługa błędów – API: typ wyliczeniowy `cudaError`

```
enum cudaError {
   cudaSuccess = 0,
   cudaErrorMissingConfiguration,
   cudaErrorMemoryAllocation,
   cudaErrorInitializationError,
   cudaErrorLaunchFailure,
   cudaErrorPriorLaunchFailure,
   cudaErrorLaunchTimeout,
   cudaErrorLaunchOutOfResources,
   cudaErrorInvalidDeviceFunction,
   cudaErrorInvalidConfiguration,
   cudaErrorInvalidDevice,
   cudaErrorInvalidValue,
   cudaErrorInvalidPitchValue,
   cudaErrorInvalidSymbol,
   cudaErrorMapBufferObjectFailed,
   :
```

Obsługa błędów – API: typ wyliczeniowy `cudaError`

```
:
cudaErrorUnmapBufferObjectFailed,
cudaErrorInvalidHostPointer,
cudaErrorInvalidDevicePointer,
cudaErrorInvalidTexture,
cudaErrorInvalidTextureBinding,
cudaErrorInvalidChannelDescriptor,
cudaErrorInvalidMemcpyDirection,
CudaErrorAddressOfConstant,
CudaErrorTextureFetchFailed,
CudaErrorTextureNotBound,
CudaErrorSynchronizationError,
CudaErrorInvalidFilterSetting,
CudaErrorInvalidNormSetting,
CudaErrorMixedDeviceExecution,
CudaErrorCudartUnloading,
CudaErrorUnknown,
:
```

Obsługa błędów – API: typ wyliczeniowy `cudaError`

```
    :
    cudaErrorNotYetImplemented,
    cudaErrorMemoryValueTooLarge,
    cudaErrorInvalidResourceHandle,
    cudaErrorNotReady,
    cudaErrorInsufficientDriver,
    cudaErrorSetOnActiveProcess,
    cudaErrorNoDevice,
    cudaErrorStartupFailure,
    cudaErrorApiFailureBase
};
```

Obsługa błędów – API: typ `cudaError_t`

```
typedef enum cudaError cudaError_t;
```

Obsługa błędów – API: pobranie kodu błędu

```
cudaError_t cudaGetLastError (void)
```

Funkcja odczytuje kod błędu ewentualnie wygenerowany w trakcie którejś z poprzednio zainicjowanych operacji i **zeruje go** (!)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- każda inna wartość – kod błędu

Obsługa błędów – API: pobranie opisu błędu

```
const char *cudaGetErrorString(
   cudaError_t    error
)
```

Funkcja zwraca wskaźnik na łańcuch zawierajacy opis błędu odpowiadający przekazanemu do funkcji kodowi błędu

Wynik:

- wskaźnika na łańcuch ASCIZZ

Zdarzenia – API : ustawienie flag urządzenia

```
cudaError_t cudaSetDeviceFlags(
 int  flags
 )
```

Parametry:

- flags – maska bitowa flag ustanawiających różne aspekty konfiguracji urządzenia CUDA; na razie interesująca jest dla nas ta, którą reprezentuje stała `cudaDeviceMapHost`

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorInvalidDevice`, `cudaErrorSetOnActiveProcess` - błąd

Pamięć mapowana – przykład

część 1/5

```
#include <stdio.h>
#include <assert.h>

// definiujemy rozmiar problemu i rozmiar bloku
#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

// funkcja powiększa o jeden każdy element tablicy a przechowywanej w
// pamięci hosta
void incrementArrayOnHost(float *a, int N)
{
  int i;
  for (i=0; i < N; i++)
       a[i] = a[i]+1.f;
}

// kernel powiększa o jeden element tablicy przypadający na
// wykonujący go wątek
__global__ void incrementArrayOnDevice(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N)
       a[idx] = a[idx]+1.f;
}
```

Pamięć mapowana – przykład

część 2/5

```
// funkcja pomocnicza (dla miłośników programowania defensywnego)
void checkCUDAError(const char *msg)
{
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString(err) );
    exit(EXIT_FAILURE);
  }
}
```

Pamięć mapowana – przykład
część 3/5

```
int main(void)
{
  float *a_m; // wskaźnik do pamięci hosta
  float *a_d; // wskaźnik do zamapowanej pamięci urządzenia
  float *check_h; // wskaźnik do pamięci hosta (użyty do sprawdzenia wyniku)
  int i, N = NUMBER_OF_ARRAY_ELEMENTS;
  size_t size = N * sizeof(float);
  cudaDeviceProp deviceProp;

#if CUDART_VERSION < 2020
#error "To urzadzenie nie wspiera mapowania pamieci ;(\n"
#endif

  // Pobierz własności i sprawdź, czy urządzenie #0 wspiera mapowanie
  cudaGetDeviceProperties(&deviceProp, 0);
  checkCUDAError("cudaGetDeviceProperties");

  if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Urzadzenie %d nie wspiera mapowania pamieci ;(\n", 0);
    exit(EXIT_FAILURE);
  }
:
:
:
```

Pamięć mapowana – przykład

część 4/5

```
:
:
// przygotuj urządzenie do mapowania pamięci
cudaSetDeviceFlags(cudaDeviceMapHost);
checkCUDAError("cudaSetDeviceFlags");

// przydziel pamięć mapowaną
cudaHostAlloc((void**)&a_m, size, cudaHostAllocMapped);
checkCUDAError("cudaHostAllocMapped");

// pobierz wskaźnik na pamięć dzielona użyteczny dla urządzenia
cudaHostGetDevicePointer((void**)&a_d, (void*)a_m, 0);
checkCUDAError("cudaHostGetDevicePointer");

// inicjacja danych hosta
for (i=0; i<N; i++)
     a_m[i] = (float)i;

// przygotowanie konfiguracji dla dla odpalenia wątków
int blockSize = N_THREADS_PER_BLOCK;
int nBlocks = N / blockSize + (N % blockSize > 0 ? 1 : 0);
:
:
```

Pamięć mapowana – przykład

część 5/5

```
  :
  :
  // odpalenie wątku
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
  checkCUDAError("incrementArrayOnDevice");

  // dla pewności zrobimy to samo używając CPU hosta
  check_h = (float *)malloc(size);
  for (i=0; i<N; i++)
       check_h[i] = (float)i;
  incrementArrayOnHost(check_h, N);

  // upewniamy się, że wszystkie wątki sią zakończyły
  cudaThreadSynchronize();
  checkCUDAError("cudaThreadSynchronize");

  // sprawdzamy poprawność obliczeń
  for (i=0; i<N; i++)
       assert(check_h[i] == a_m[i]);

  // sprzątamy
  free(check_h); // zwalniamy pamięć hosta
  cudaFreeHost(a_m); // zwalniamy pamięć dzieloną
  return 0;
}
```

CUDA
Praca asynchroniczna

Praca asynchroniczna w środowisku CUDA:

- pracą asynchroniczną nazywamy sytuację, kiedy kod urządzenia wykonuje się równolegle z kodem hosta w sposób niezależny
- pracę asynchroniczną umożliwia fakt, że niektóre z funkcji środowiska CUDA „wracają” do kodu hosta zanim jeszcze zakończy się zadanie, jakie zainicjowały
- asynchroniczne są:
  - odpalenia kerneli
  - funkcje transferu danych pomiędzy pamięcią hosta i urządzenia sufiksowane frazą **Async**
  - funkcje transferu danych urządzenie ↔ urządzenie
  - funkcje wypełniania pamięci

Praca asynchroniczna w środowisku CUDA:

- możliwe jest globalne zablokowanie asynchronicznego odpalania kerneli poprzez zdefiniowanie zmiennej środowiskowej

  ```
  CUDA_LAUNCH_BLOCKING=1
  ```

- rozwiązanie to jest przeznaczone wyłącznie do wykorzystania w trakcie debugowania kodu i nie powinno być używana w kodzie produkcyjnym

Nakładanie transferu danych na wykonanie kerneli:

- niektóre urządzenia z CC>=1.1 mogą transferować dane pomiędzy pamięcią „page-locked” i pamięcią urządzenia w trakcie pracy kodu kernela
- aplikacja może sprawdzić dostępność tej opcji poprzez wykonanie funkcji **`cudaGetDeviceProperties()`** i sprawdzenie wartości pola **`deviceOverlap`**
- możliwość ta jest w obecnych urządzeniach wspierana tylko w przypadku transferów nie dotyczących pamięci przydzielonej przez funkcję **`cudaMallocPitch()`**

Nakładanie wykonań kerneli:

- niektóre urządzenia z `CC>=2.0` mogą równolegle wykonywać więcej niż jeden kod kernela równocześnie
- aplikacja może sprawdzić dostępność tej opcji poprzez wykonanie funkcji **`cudaGetDeviceProperties()`** i sprawdzenie wartości pola **`concurrentKernels`**
- maksymalna dopuszczalna liczba równolegle wykonujących się kerneli wynosi 4

Nakładanie transferów danych:

- niektóre urządzenia z `CC>=2.0` mogą równolegle kopiować dane z pamięci „page-locked” hosta do urządzenia i z pamięci urządzenia do pamięci „page-locked” hosta

Strumienie (streams):

- strumienie są środkiem umożliwiającym aplikacjom synchronizowanie operacji przebiegających asynchronicznie (*wbrew pierwszemu wrażeniu zdanie to nie jest masłem maślanym*) :)
- strumień jest zbiorem poleceń wykonywanym w określonej kolejności
- różne strumienie mogą wykonywać swoje polecenia w dowolnym porządku honorując się wzajemnie bądź całkowicie równolegle
- żadne z powyższych zachowań nie jest gwarantowane i w konkretnym przypadku każde z nich może wystąpić z równym prawdopodobieństwem

Strumienie:

- strumień definiuje się poprzez wykreowanie obiektu strumienia i przekazanie go jako parametru do uporządkowanego zbioru odpaleń kernela i/lub transferów danych host ↔ urządzenie
- kod w poniższym przykładzie tworzy dwa strumienie i przydziela pamięć w bloku pamięci „page-locked” hosta:

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
   cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

teraz każdy z tych strumieni jest definiowany jako złożenie następujacych operacji:

- jedno kopiowanie danych host → urządzenie
- jedno odpalenie kernela
- jedno kopiowanie danych urządzenie → host

```
for (int i = 0; i < 2; ++i)
   CudaMemcpyAsync(inputDevPtr + i * size,
                   hostPtr + i * size,
                   size, cudaMemcpyHostToDevice,stream[i]);
for (int i = 0; i < 2; ++i)
   MyKernel<<<100, 512, 0, stream[i]>>>
   (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
   cudaMemcpyAsync(hostPtr + i * size,
                   outputDevPtr + i * size,size,
                   cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

UWAGA!

Asynchroniczne wykonanie obu strumieni nastąpi tylko wtedy, gdy wszystkie transfery odbywające się w ich ramach dotyczą pamięci „page-locked”

Strumienie:

- wykonanie z użyciem strumieni musi kończyć się wywołaniem funkcji `cudaThreadSynchronize()`
- wywołanie to wymusza na środowisku zatrzymanie wykonywania kodu wykonywanego poza strumieniami do chwili zakończenia sekwencji poleceń we wszystkich strumieniach
- funkcja `cudaStreamSynchronize()` wymusza zatrzymanie środowiska do chwili zakończenia wszystkich wcześniej zainicjowanych w strumieniu czynności
- funkcja `cudaStreamQuery()` pozwala odpytać strumień o to, czy jego sekwencja została zakończona

Strumienie:

- strumień jest unicestwiany poprzez wywołanie funkcji **`cudaStreamDestroy()`**
- funkcja ta czeka na zakończenie wszystkich operacji strumienia, a następnie niszczy strumień i oddaje sterowanie do hosta

```
for (int i = 0; i < 2; ++i)
   cudaStreamDestroy(stream[i]);
```

Strumienie:

- dwa polecenia z różnych strumieni nie mogą być wykonane równolegle, jeśli co najmniej jedna z poniższych operacji jest inicjowana przez dowolny wątek hosta w trakcie pracy strumienia:
  - przydział pamięci „page-locked”
  - przydział pamięci urządzenia
  - kopiowanie pamięci urządzenie ↔ urządzenie

Strumienie – API : typ cudaStream_t

```
typedef int cudaStream_t;
```

Strumienie – API : wytworzenie obiektu strumienia

```
cudaError_t cudaStreamCreate  (
   cudaStream_t *   pStream
   )
```

Parametry:

- pStream – wskaźnik na daną reprezentującą strumień (funkcja wykreuje jej wartość)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- cudaErrorInvalidValue - błąd

Strumienie – API : unicestwienie obiektu strumienia

```
cudaError_t cudaStreamDestroy (
   cudaStream_t   stream
   )
```

Parametry:

- stream – dana reprezentująca strumień

Wynik:

- `cudaSuccess` – wykonanie poprawne
- cudaErrorInvalidResourceHandle - błąd

Strumienie – API : odpytanie obiektu strumienia

```
cudaError_t cudaStreamQuery(
   cudaStream_t   stream
   )
```

Parametry:

- stream – dana reprezentująca strumień

Wynik:

- `cudaSuccess` – wszystkie wcześniejsze operacje we wskazanym strumieniu są już zakończone
- cudaErrorNotReady – strumień ciągle wykonuje swój ciąg operacji
- cudaErrorInvalidResourceHandle - błąd

Strumienie – API : zatrzymanie do chwili ukończenia operacji w strumieniu

```
cudaError_t cudaStreamSynchronize(
   cudaStream_t   stream
)
```

Parametry:

- stream – dana reprezentująca strumień

Wynik:

- `cudaSuccess` – wykonanie poprawne
- cudaErrorInvalidResourceHandle - błąd

Zdarzenia (events):

- zdarzenia są środkiem umożliwiającym precyzyjne mierzenie interwałów czasu jakie zachodzą pomiędzy wyróżnionymi punktami kodu
- zdarzenie jest rejestrowane dokładnie wtedy, kiedy zostały zakończone wszystkie poprzedzające je operacje (dotyczy to także operacji wykonywanych w strumieniu)
- do utworzenia obiektu zdarzenia wykorzystuje się funkcję **`cudaEventCreate()`**

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Zdarzenia:

- do „uchwycenia” używa się funkcji **`cudaEventRecord()`**, a do wyliczenia czasu jaki upłynął między dwoma zdarzeniami – funkcji **`cudaEventElapsedTime()`**

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
       MyKernel<<<100, 512, 0, stream[i]>>>
       (outputDev + i * size, inputDev + i * size, size);
for (int i = 0; i < 2; ++i)
       cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                       size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Zdarzenia :

- po użyciu obiekty zdarzeń powinny zostać unicestwione

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Zdarzenia – API : typ `cudaEvent_t`

- `typedef int cudaEvent_t;`

Zdarzenia – API : utworzenie obiektu zdarzenia

```
cudaError_t cudaEventCreate(
  cudaEvent_t *event
)
```

Parametry:

- `event` - na tworzony obiekt (*wartość ustawiana przez funkcję*)

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorInitializationError`, `cudaErrorPriorLaunchFailure`, `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation` - błąd

Zdarzenia – API : uchwycenie zdarzenia

```
cudaError_t cudaEventRecord(
  cudaEvent_t    event,
  cudaStream_t   stream )
```

Parametry:

- event - zdarzenie do uchwycenia
- stream - strumień, w którym zdarzenie ma być uchwycone

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorInvalidValue`, `cudaErrorInitializationError`, `cudaErrorPriorLaunchFailure`, `cudaErrorInvalidResourceHandle` - błąd

Zdarzenia – API : zatrzymanie do momentu rzeczywistego uchwycenia zdarzenia

```
cudaError_t cudaEventSynchronize(
    cudaEvent_t event
)
```

Parametry:

- event - zdarzenie na uchwycenie którego należy poczekać

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaErrorInitializationError`, `cudaErrorPriorLaunchFailure`, `cudaErrorInvalidValue` `cudaErrorInvalidResourceHandle` - błąd

Zdarzenia – API : obliczenie czasu między zdarzeniami

```
cudaError_t cudaEventElapsedTime(
   float          *ms,
   cudaEvent_t    start,
   cudaEvent_t    end
)
```

Parametry:

- ms - wyliczony czas (w milisekundach)
- start – zdarzenie początkowe
- stop – zdarzenie końcowe

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventRecord` - błąd

Zdarzenia – API : unicestwienie obiektu zdarzenia

```
cudaError_t cudaEventDestroy(
   cudaEvent_t    event
   )
```

Parametry:

- event – zdarzenie do likwidacji

Wynik:

- `cudaSuccess` – wykonanie poprawne
- `cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventRecord, cudaEventElapsedTime` - błąd