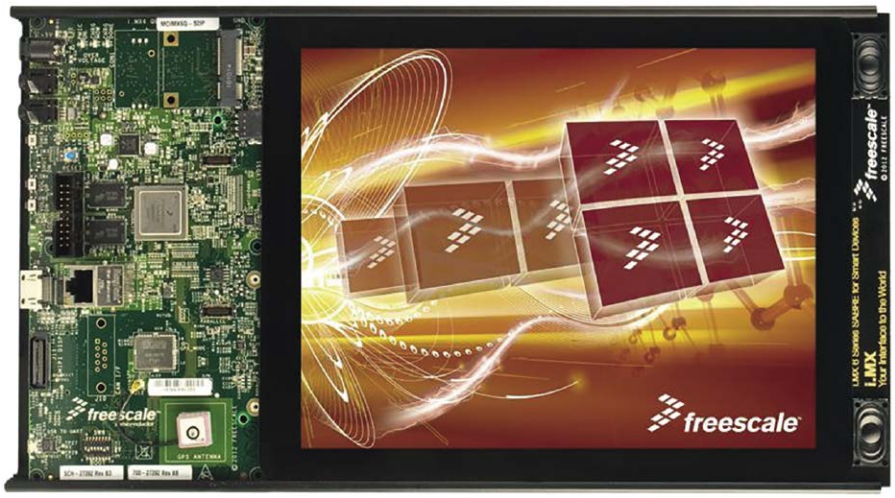
Jednostka GPU oraz OpenCL – aplikacje

W artykule zaprezentowano przykładową aplikacją – symulowanie sił oddziaływujących na cząsteczki oraz animowanie ich ruchu na ekranie zestawu ewaluacyjnego i.MX6 Sabre. Omówiono również ramy aplikacji OpenCL, typy zmiennych mające zastosowanie, sposób funkcjonowania aplikacji (podział na Hosta i Kernel), który jest nieco odmienny od typowego oraz pokazano listingi z zaznaczeniem i omówieniem ważniejszych części kodu źródłowego.



Fotografia 1. Platforma Sabre wyposażona w procesor i.MX 6

Jak wspomniano w pierwszej części artykułu (EP 6/2015), aplikacja OpenCL składa się z dwóch komponentów: jeden jest uruchamiany na Hoście (służy on do zarządzania systemem i zawiera kod kontrolujący wykonywanie aplikacji), natomiast drugi jest uruchamiany na Urządzeniu Przetwarzającym, zwykle wykonującym najcięższą pracę przy przetwarzaniu danych. W niektórych urządzeniach (na przykład – w typowym komputerze PC wyposażonym w kartę graficzną) są to odseparowane urządzenia, ale nie w systemach embedded, ponieważ większość nowoczesnych procesorów aplikacyjnych zawiera zintegrowaną jednostkę GPU, które może wykonywać aplikacje OpenCL.

Dla potrzeb aplikacji demonstracyjnej wykorzystaliśmy ten sam 4-rdzeniowy procesor i.MX6, który opisywaliśmy w pierwszej części artykułu. Tam posłużyliśmy się nim dla celu zaprezentowania struktury nowoczesnego GPU widzianego z perspektywy OpenCL. Host jest reprezentowany przez klaster 4 rdzeni ARM A9, natomiast Urządzenie Przetwarzające przez procesor graficzny GC2000. Jako system do uruchomienia naszej aplikacji wybraliśmy płytkę demonstracyjną i.MX6 Sabre, jedną z platform referencyjnych dla tego układu SoC (pokazano ją na **fotografii 1**).

Tworząc może nieco mało ambitną aplikację OpenCL typu „Hello World" na ekranie platformy Sabre wyświetlimy symulację przemieszczania się cząstek poddanych (w systemie zamkniętym) oddziaływaniu predefiniowanych sił. Początkowy zestaw cząstek będzie określony za pomocą bitmapy. Na **rysunku 2** pokazano wygląd ekranu w różnych stanach pracy aplikacji (lub czasu życia systemu cząstek). Mamy nadzieję, że ta przykładowa aplikacji zachęci czytelnika do jej powielenia, samodzielnego eksperymentowania, a także przeanalizowania znaczenia i sposobu działania poszczególnych funkcji.

Dla ułatwienia samodzielnej pracy, omówimy niezbędne nastawy systemu, rolę Hosta oraz Urządzenia Przetwarzającego.

Nastawy sytemu

Jako środowiska projektowego dla Hosta użyjemy kompilatora języka C pracującego pod kontrolą Linuksa. Dla naszej przykładowej platformy Sablre użyjemy najnowszej wersji Linuksa BSP, dostępnego do pobrania na stronie *http://www.freescale.com* w menu *Software and Tools/Software development tools* dla procesora i.MX6. W czasie pisania artykułu najnowsza wersja Linuksa BSP nosi oznaczenie L3_10_53_1.1.0. Sposób wykonania obrazu systemu – oczywiście po pobraniu Linuksa BSPA – jest opisany w *Freescale Yocto Project User's Guide*. Dla potrzeb aplikacji demonstracyjnej OpenCL będziemy używali bufora ramek obrazu, więc najlepiej posłużyć się obrazem o nazwie **fsl-image-gui**, ponieważ zawiera on wszystkie potrzebne biblioteki wymagane przez naszą aplikację.

Jeśli program demonstracyjny będzie uruchamiany na innej platformie z i.MX6, to należy włączyć wsparcie dla obsługi bufora ramek oraz mieć zainstalowany pakiet **gpu-viv-bin-mx6q**.

Kompletny kod źródłowy opisywanej aplikacji oraz instrukcja jej skompilowania i uruchomienia/wgrania na platformie docelowej są opublikowane pod adresem internetowym społeczności Freescale *https://community.freescale.com/docs/DOC-103684*.

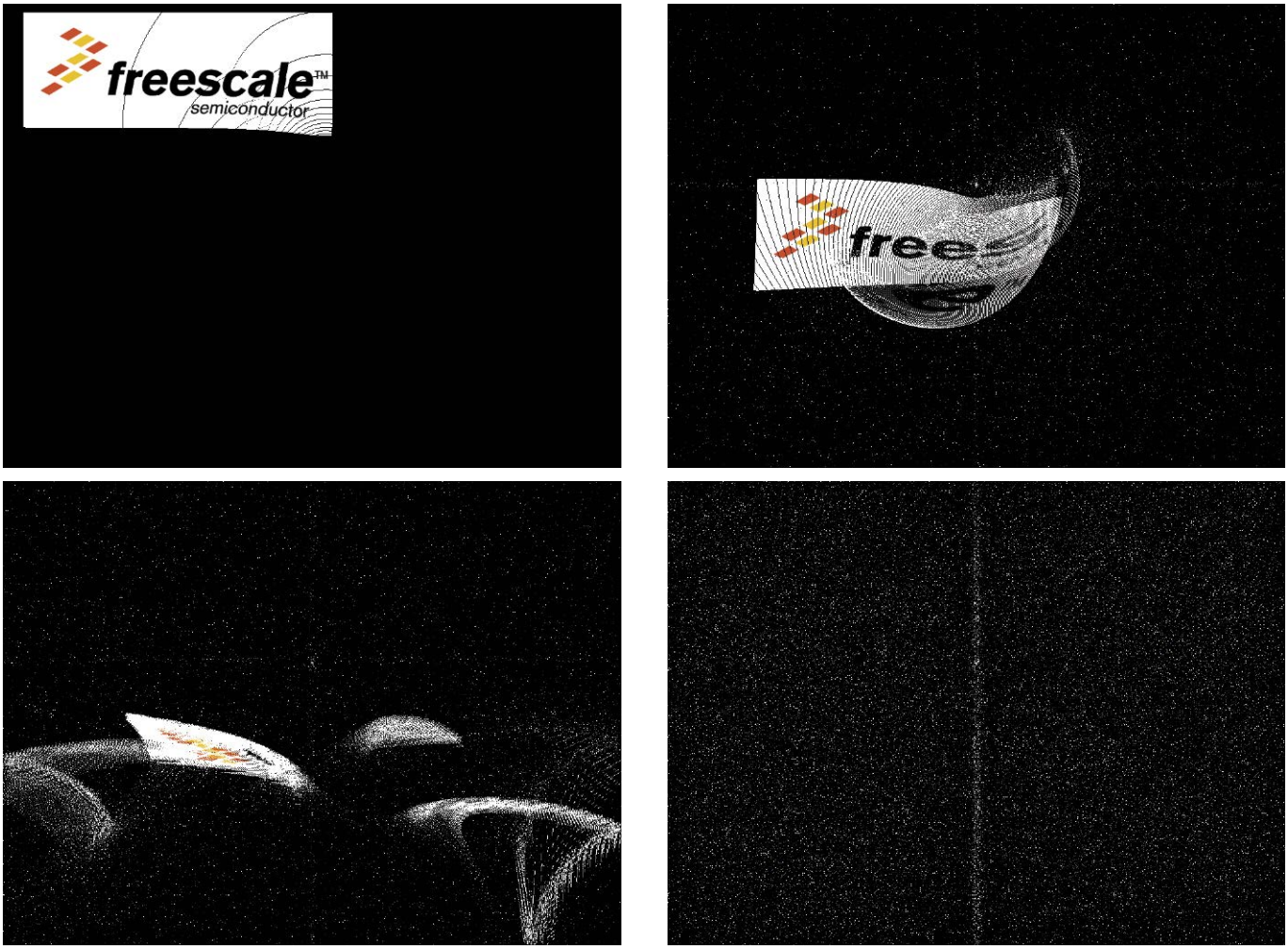
Opis aplikacji dla Hosta

Zanim zaprezentujemy szczegóły dotyczące wykonania aplikacji dla Hosta, pokrótce omówimy funkcje pełnione przez poszczególne komponenty systemu OpenCL:

- Interfejs użytkownika.
- Konfiguracja systemu, wliczając w to rozpoznanie i konfigurowanie Urządzenia Przetwarzającego.
- Utworzenie kanałów komunikacyjnych służących do wymiany danych i komend z Urządzeniem Przetwarzającym.
- Transmisja danych, przetwarzanie, gromadzenie i prezentowanie rezultatów przetwarzania wykonywanego przez Urządzenie Przetwarzające.

Zanim zidentyfikujemy i wyjaśnimy sposób, w jaki są te zadania realizowane, pokrótce zaprezentujemy minimalny zestaw struktur danych OpenCL, dzięki którym jest możliwe wykonanie wszystkich wymienionych zadań:

- **Kontekst** (*cl_context*):
  - Program obsługi zapewniany przez framework OpenCL na żądanie dostępu przez Hosta do Urządzenia Przetwarzającego.
  - Pozwala frameworkowi OpenC, a w naszym wypadku – sterownikowi



Rysunek 2. Obraz wyświetlany przez aplikację demonstracyjną na ekranie LCD

```
187  /* Initializatio sequence for the OpenCL environment */
188  err = cl_create(&clctx);
189  if (err) goto error;
190
191  err = cl_alloc(clctx, &vars, &off, forces, nbelem);
192  if (err) goto error;
193
194  /* Read image, allocates and set particle=pixel */
195  err = readimage(imgname, vars, fixs, &nbelem);
196
197  /* Clear the screen */
198  memset(pfb, 0, fbsize);
199  ppfb = (uint16 *) pfb;
200
201  err = cl_init(clctx, nbelem);
202  if (err) goto error;
203
204  /* main loop for the application */
205  while(1)
206  {
207      err = cl_run(clctx, &off, nbelem);
208      if (err) goto error;
209      /* Wait for the vsync to avoid tearing */
210      ioctl(fdfb, FBIO_WAITFORVSYNC, NULL);
211      /* Clear the screen */
212      memset(pfb, 0, fbsize);
213      /* Plot elements */
214      for (i = 0; i < nbelem; i++)
215      {
216          ppfb[off[i]] = fixs[i].rgb;
217      }
218  }
219
```

W pierwszym kroku identyfikujemy Urządzenie Przetwarzające, pobieramy kontekst i inne związane z nim dane, które będą wykorzystane przez aplikację definiującą clctxt.

Przydzielamy pamięć wymaganą do przechowywania danych wejściowych (współrzędne cząstek w vars) i informacji wyjściowej z Urządzenia Przetwarzającego po przetworzeniu danych (zostanie zapamiętana w off). Pamięć będzie przydzielana przez OpenCL na podstawie wielkości danych wejściowych (nbelem), a Host pobierze potrzebne wskaźniki do danych. Siły, które będą oddziaływały na cząstki są rozmieszczone statycznie, zainicjowane przez Hosta i przekazane do Urządzenia Przetwarzającego w forces.

Czytamy graficzny plik wejściowy. Współrzędne cząstek, które będą transformowane przez Urządzenie Przetwarzające są odczytywane bezpośrednio z vars. Obiekty w pamięci są alokowane przez OpenCL i służą do inicjowania Kernela GPU. Kolor będzie zapamiętany w fixis.

Funkcja kompiluje zbiór źródłowy implementujący Kernel uruchamiany w GPU i ładuje Kernel do GPU w celu uruchomienia.

Uruchomienie Kernela. Gdy funkcja zostanie zakończona, w zmiennej off zostaną zwrócone efekty wykonanego przez nią przetwarzania.

Funkcja wyświetla rezultat na ekranie dostępnym poprzez mapowany w pamięci bufor ramki wskazywany przez ppfb. Zmienna off przechowuje nowe koordynaty cząstki (po przetworzeniu przez Kernel uruchomiony na GPU), a fixis przechowuje informację o kolorze.

Rysunek 3. Szczegóły implementacji *main()*

GPU, na zarządzenie odpowiednimi obiektami (pamięć, komendy, kernel, synchronizacja) odnoszącymi się do interakcji pomiędzy Hostem a Urządzeniem Przetwarzającym.

- **Kolejka komend** (*cl_command_queue*):
  - Kanał komunikacyjny pomiędzy Hostem i kontekstem skojarzonym z danym Urządzeniem Przetwarzającym. Pozwala on Hostowi na żądanie wykonywania zadań związanych z obiektami skojarzonymi z kontekstem.
  - Użycie kolejki pozwala na łatwe synchronizowanie żądanych operacji – najwygodniejsze jest żądanie wykonania zadań w ramach frameworku Open CL/Urządzenia Przetwarzającego, co może mieć wpływ na wydajność.
  - Dozwolone jest powiązanie wielu kolejek z tym samym kontekstem, ale w takim wypadku synchronizacja musi być zapewniana przez Hosta. Ta technika jest używana w celu zapewnienia optymalnej wydajności zwłaszcza wtedy, gdy jest wiele Urządzeń Przetwarzających.

```
29  int cl_create(void **ctx)
30  {
31      cl_int err = CL_SUCCESS;
32      struct cl_context *clctx = (struct cl_context *) malloc(sizeof(struct cl_context));
33      if(clctx == NULL) goto error;
34      *ctx = clctx;
35
36      /* Get an OpenCL platform */
37      cl_platform_id cpPlatform;
38      err = clGetPlatformIDs(1, &cpPlatform, NULL);
39      if (err != CL_SUCCESS) goto error;
40      /* Get a GPU device */
41      err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &clctx->cdDevice, NULL);
42      if (err != CL_SUCCESS) goto error;
43      /* Create a context to run OpenCL enabled GPU */
44      clctx->GPUContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, &err);
45      if (err != CL_SUCCESS) goto error;
46      /* Create a command-queue on the GPU device */
47      clctx->cqCommandQueue = clCreateCommandQueue(clctx->GPUContext, clctx->cdDevice, 0, &err);
48      if (err != CL_SUCCESS) goto error;
49
50      return 0;
```

Pobranie identyfikatora GPU, które będzie używane. W związku z tym, że w naszym układzie SoC mamy tylko jedno GPU, łatwo jest „dostać się" do niego. Jeśli byłoby obecne więcej GPU, wymagane byłoby ich parsowanie i selekcja z listy zwróconej przez funkcję.

Tworzenie kolejki komend, które będą używane. Tworzymy ją w kolejności wykonywania – trzeci parametr wywołania funkcji pozwala na żądanie wykonania komendy poza kolejką, jeśli to konieczne.

Rysunek 4. Szczegóły funkcji *cl_create()*

```
60  int cl_alloc(void *ctx, struct varbody **vars, cl_uint **offsets, struct force *forces, int nbelem)
61  {
62      cl_int err = CL_SUCCESS;
63      struct cl_context *clctx = ctx;
64      /* Allocate memory for the initial set of particles */
65      clctx->GPUvars = clCreateBuffer(clctx->GPUContext, CL_MEM_READ_WRITE |
66              CL_MEM_ALLOC_HOST_PTR, sizeof(struct varbody) * nbelem, NULL, &err);
67      if (err != CL_SUCCESS) goto error;
68      /* Allocate memory for the output buffer containing the processed set of particles */
69      clctx->GPUoffsets = clCreateBuffer(clctx->GPUContext, CL_MEM_WRITE_ONLY |
70              CL_MEM_ALLOC_HOST_PTR, sizeof(cl_uint) * nbelem, NULL, &err);
71      if (err != CL_SUCCESS) goto error;
72      /* Allocate memory for the set of force vectors to act on the particles */
73      clctx->GPUforc = clCreateBuffer(clctx->GPUContext, CL_MEM_READ_ONLY |
74              CL_MEM_COPY_HOST_PTR, sizeof(struct force) * MAX_FORCES, forces, &err);
75      if (err != CL_SUCCESS) goto error;
76      /* Map the buffers and copy the address such that the application can access these */
77      *vars = clEnqueueMapBuffer(clctx->cqCommandQueue, clctx->GPUvars, CL_TRUE, CL_MAP_WRITE,\
78       0, sizeof(struct varbody) * nbelem, 0, NULL, NULL, &err);
79      if (err != CL_SUCCESS) goto error;
80
81      *offsets = clEnqueueMapBuffer(clctx->cqCommandQueue, clctx->GPUoffsets, CL_TRUE, CL_MAP_READ,\
82       0, sizeof(cl_uint) * nbelem, 0, NULL, NULL, &err);
83      if (err != CL_SUCCESS) goto error;
84
85      return 0;
```

Żądanie utworzenia przez OpenCL obiektu w pamięci (GPUvars) zapamiętanego w kontekście lokalnym, odpowiednim dla Urządzenia Przetwarzającego, dostępnym dla Hosta jako vars (patrz niżej). Za pomocą drugiego parametru określamy typ dostępu (CL_MEM_READ_WRITE) z punktu widzenia Urządzenia Przetwarzającego do przydzielonego obszaru pamięci (MEM_ALLLOC_HOST_PTR).

Tworzenie obiektu w pamięci – tym razem do skopiowania danych alokowanych przez Hosta dla Urządzenia Przetwarzającego. Następnie podajemy CL_MEM_READ_ONLY i CL_MEM_COPY_HOST_PTR.

Mapowanie utworzonego wcześniej w pamięci obiektu (GPUvars) w przestrzeni adresowej Hosta (vars). Dla uproszczenia, wywołanie jest synchroniczne (CL_TRUE) – Host zapisze w pamięci obiekt (CL_MAP_WRITE).

Rysunek 5. Szczegóły implementacji funkcji *cl_alloc()*

```
105     /* Find and read the kernel source code, then load the program object in the GPU context. */
106     fd = fopen("physics.cl", "r");
107     if (fd == NULL)
108     {
109         printf("\n Cannot open CL kernel source file");
110         goto error;
111     }
112     fseek(fd, 0, SEEK_END);
113     srcsize = ftell(fd);
114     srcstr[0] = malloc(srcsize+1);
115     fseek(fd, 0, SEEK_SET);
116     if(1 != fread(srcstr[0], srcsize, 1, fd))
117     {
118         printf("\n Cannot read CL kernel source");
119         goto error;
120     }
121     srcstr[0][srcsize] = '\0';
122     clctx->OpenCLProgram = clCreateProgramWithSource(clctx->GPUContext, 1, (const char **)srcstr, NULL, &err);
123     if (err != CL_SUCCESS) goto error;
124
125     /* Build the program (OpenCL JIT compilation) */
126     err = clBuildProgram(clctx->OpenCLProgram, 0, NULL, NULL, NULL, NULL);
127     if (err != CL_SUCCESS)
128     {
129         char log[16*1024];
130         size_t logsize = 0;
131         clGetProgramBuildInfo(clctx->OpenCLProgram, clctx->cdDevice, CL_PROGRAM_BUILD_LOG, (16*1024), log, &logsize);
132         printf("\n CL COMPILER ERROR. \n ***CL_LOG_START*** \n %s \n ***CL_LOG_END*** \n", log);
133         clReleaseProgram(clctx->OpenCLProgram);
134         goto error;
135     }
136
137     /* Get a handle to the compiled OpenCL function (Kernel) */
138     clctx->CLphysic = clCreateKernel(clctx->OpenCLProgram, "Physic", &err);
139     if (err != CL_SUCCESS) goto error;
140
141     /* Associate the GPU memory objects we allocated with the Kernel arguments */
142     err = clSetKernelArg(clctx->CLphysic, 0, sizeof(cl_mem), (void*)&(clctx->GPUvars));
143     if (err != CL_SUCCESS) goto error;
144     err = clSetKernelArg(clctx->CLphysic, 1, sizeof(cl_mem), (void*)&(clctx->GPUforc));
145     if (err != CL_SUCCESS) goto error;
146     err = clSetKernelArg(clctx->CLphysic, 2, sizeof(cl_mem), (void*)&(clctx->GPUoffsets));
147     if (err != CL_SUCCESS) goto error;
```

Funkcja Kernela, która będzie wykonywana przez GPU, jest zapamiętana w zbiorze physiscs.cl. Musimy odczytać go z aplikacji Hosta. Zbiór jest spodziewany i powinien być zapisany w tym samym katalogu, z którego uruchomiono wykonywanie aplikacji Hosta.

Ładuje kod źródłowy dla Kernela do obiektu cl_program – reprezentacji Kernela utworzonej przez framework OpenCL służącej do kompilowania i linkowania Kernela.

Synchronicznie kompiluje i tworzy Kernel. Powstałe błędy są natychmiast odczytywane i wyświetlane w kolejnej sekcji.

Ostatni krok w procesie przygotowywania Kernela OpenCL do uruchomienia – linkowanie obiektów z pamięci, które są połączone z aplikacją Hosta. Te obiekty będą dostępne dla Kernela jako numerowane parametry w buforze w pamięci – specyfikację struktury danych podano odrębnie.

Rysunek 6. Szczegóły implementacji funkcji *cl_init()*

- **Kernel** (*cl_kernel*):
  - Funkcja przeznaczona do uruchomienia na Urządzeniu Przetwarzającym.
  - Jest kompilowana (w trybie *off line* lub podczas pracy) i ładowana do urządzenia przetwarzającego przez Hosta.
- **Obiekty w pamięci** (*cl_mem*):
  - Mogą służyć do jednokierunkowej lub dwukierunkowej transmisji danych pomiędzy Hostem i Urządzeniem Przetwarzającym.
  - Najprostszą postacią jest tablica niezwymiarowana, ale używane mogą być również tablice wielowymiarowe (do 3 wymiarów).
  - W aktualnej implementacji wymiany danych są dwie najważniejsze metody realizacji tego zadania:
    * Mapowanie fizycznie dzielonych obszarów pamięci w obu kontekstach wykonywania zadań (Host i Urządzenie Przetwarzające).
    * Poleganie na kopiach danych wykonywanych w razie potrzeby przez framework OpenCL.
  - Wybór metody transmisji/wymiany danych ma znaczny wpływ na wydajność i jest silnie związany z systemem, pod którego kontrolą jest uruchamiana aplikacja.

W naszym przykładzie oprogramowanie Hosta jest implementowane w dwóch zbiorach źródłowych w języku C:

- **main.c**:
  - Inicjalizuje ekran i struktury danych reprezentujące siły występujące w systemie.
  - Czyta bitmapy wejściowe, oddziela fragmenty, które będą przetwarzane.
  - Używa funkcji zaimplementowanych w **cl_implem.c** do utworzenia, inicjalizacji i zarządzania przetwarzaniem przez GPU.
- **cl_implem.c**:
  - Zawiera funkcje potrzebne do wsparcia interakcji pomiędzy Hostem i Urządzeniem Przetwarzającym, utworzone „w oderwaniu" frameworka OpenCL.

Implementację funkcji ***main()*** w pliku ***main.c*** pokazano na **rysunku 3** z dodatkowymi uwagami odnośnie do kolejności wykonywania. Byłoby idealnie, gdyby czytelnik równolegle zapoznał się z opisem w artykule oraz wyjaśnieniami w ramkach na rys. 3, odnosząc to do kodu źródłowego.

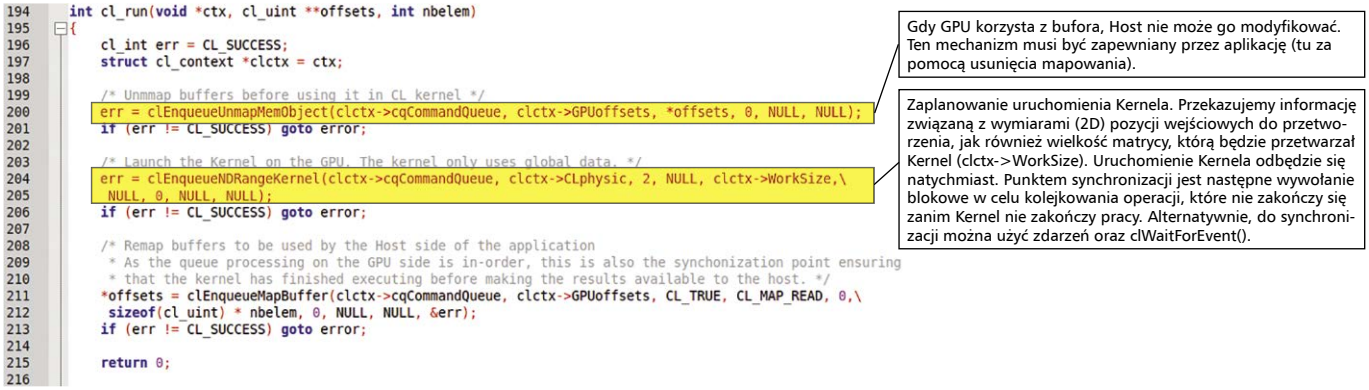
Należy zauważyć, że wersja OpenCL wspierana przez procesor aplikacyjny i.MX 6 GPU to **OpenCL 1.1** profil **embedded**. Znajduje to odzwierciedlenie w API frameworka OpenCL wykorzystanym w przykładowym programie – niektóre jego elementy zmieniły się po przeniesieniu do OpenCL 2.0.

Funkcje pomocnicze obecne w OpenCL, które są wywoływane przez funkcję *main()* są zaprezentowane na **rysunkach 4…8**. Nie będziemy komentowali wywołania każdej z funkcji OpenCL, ale opatrzymy uwagami tylko te, które pomogą czytelnikowi w łatwiejszym zrozumieniu aspektów aplikacji utworzonej za pomocą OpenCL.

Opis aplikacji dla Urządzenia Przetwarzającego

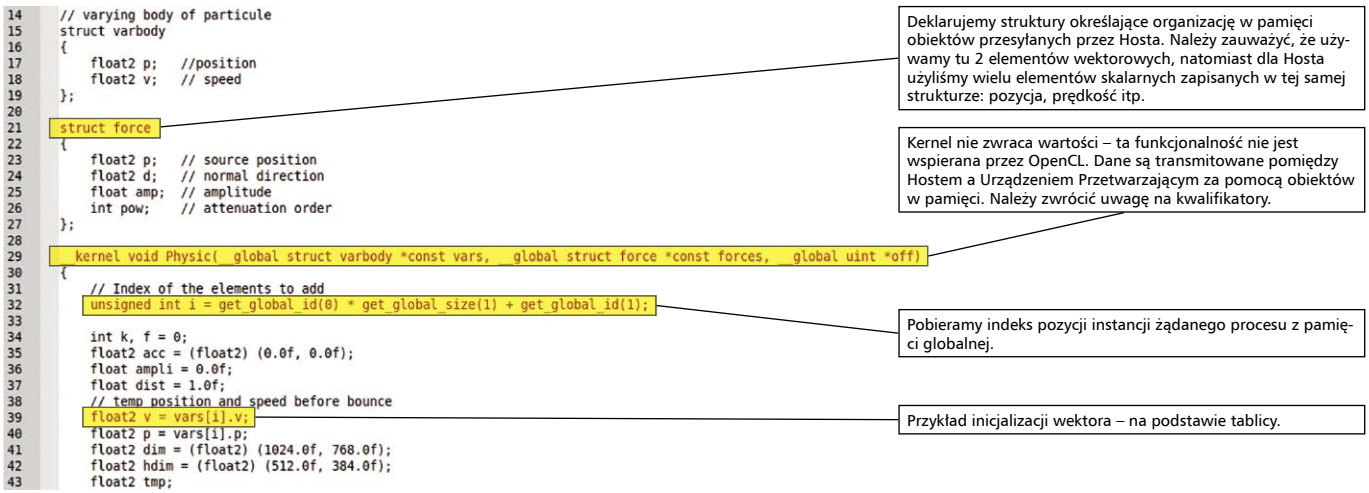
Urządzenie Przetwarzające uruchamia „kernele", które typowo są budowane (kompilowane) przez Hosta (może to być również zrobione off line) i ładowane podczas pracy systemu. Kod źródłowy Kernela jest tworzony w języku bardzo zbliżonym do ANSI C. Rzućmy okiem na minimalny zestaw zagadnień, które musi znać czytelnik przed przystąpieniem do analizowania kodu Kernela. Te zagadnienia są związane z: podziałem obszaru pamięci aplikacji, typami danych, prototypem Kernela oraz restrykcjami OpenCL w ANSI C.

W pierwszej części artykułu wprowadziliśmy pojęcie przestrzeni adresowych definiowanych przez OpenCL. Gdy przekazywaliśmy dane pomiędzy Hostem i Urządzeniem Przetwarzającym (jak dla przykładu definicje argumentów Kernela) lub gdy




```
194  int cl_run(void *ctx, cl_uint **offsets, int nbelem)
195  {
196      cl_int err = CL_SUCCESS;
197      struct cl_context *clctx = ctx;
198
199      /* Unmmap buffers before using it in CL kernel */
200      err = clEnqueueUnmapMemObject(clctx->cqCommandQueue, clctx->GPUoffsets, *offsets, 0, NULL, NULL);
201      if (err != CL_SUCCESS) goto error;
202
203      /* Launch the Kernel on the GPU. The kernel only uses global data. */
204      err = clEnqueueNDRangeKernel(clctx->cqCommandQueue, clctx->CLphysic, 2, NULL, clctx->WorkSize,\
205       NULL, 0, NULL, NULL);
206      if (err != CL_SUCCESS) goto error;
207
208      /* Remap buffers to be used by the Host side of the application
209       * As the queue processing on the GPU side is in-order, this is also the synchonization point ensuring
210       * that the kernel has finished executing before making the results available to the host. */
211      *offsets = clEnqueueMapBuffer(clctx->cqCommandQueue, clctx->GPUoffsets, CL_TRUE, CL_MAP_READ, 0,\
212       sizeof(cl_uint) * nbelem, 0, NULL, NULL, &err);
213      if (err != CL_SUCCESS) goto error;
214
215      return 0;
216
```

Rysunek 7. Szczegóły funkcji *cl_run()*




```
14  // varying body of particule
15  struct varbody
16  {
17      float2 p;   //position
18      float2 v;   // speed
19  };
20
21  struct force
22  {
23      float2 p;   // source position
24      float2 d;   // normal direction
25      float amp;  // amplitude
26      int pow;    // attenuation order
27  };
28
29  __kernel void Physic(__global struct varbody *const vars, __global struct force *const forces, __global uint *off)
30  {
31      // Index of the elements to add
32      unsigned int i = get_global_id(0) * get_global_size(1) + get_global_id(1);
33
34      int k, f = 0;
35      float2 acc = (float2) (0.0f, 0.0f);
36      float ampli = 0.0f;
37      float dist = 1.0f;
38      // temp position and speed before bounce
39      float2 v = vars[i].v;
40      float2 p = vars[i].p;
41      float2 dim = (float2) (1024.0f, 768.0f);
42      float2 hdim = (float2) (512.0f, 384.0f);
43      float2 tmp;
```

Rysunek 8. Struktury, kernel i deklarowanie zmienny prywatnych

```
45      // forces
46      for(f = 0; f < MAX_FORCES; f++)
47      {
48          if (forces[f].pow)
49          {
50              tmp = forces[f].p - vars[i].p;
51              dist = native_rsqrt(dot(tmp, tmp));
52              ampli = forces[f].amp;
53              // optim of pown(dist, forces[f].pow+1) for small power
54              for(k = 0; k < (forces[f].pow+1); k++)
55              {
56                  ampli *= dist;
57              }
58
59              acc += ampli * (forces[f].p - vars[i].p);
60          }
61          else
62          {
63              ampli = forces[f].amp;
64              acc += (float2) (ampli * forces[f].d);
65          }
66      }
67
68      // assume time quantum = 1
69      v += acc;
70      p += v;
71
72      // clamp speed
73      v = clamp(v, -hdim, hdim);
74
75      p = vars[i].p + v;
76
77      // reflection on screen walls
78      v = (p <= (0.0f, 0.0f)) ? -v: v;
79      v = (p > (dim - (1.0f, 1.0f))) ? -v: v;
80      p = vars[i].p + v;
81
82      vars[i].p = p;
83      vars[i].v = v;
84
85      off[i] = (uint)p.y * (uint)dim.x + (uint) p.x;
86  }
87
```

Aktualizujemy prędkość i przyśpieszenie cząstki przemieszczającej się pod wpływem sił zadeklarowanych w systemie. Należy zauważyć, że spośród obiektów vars przekazanych przez Hosta za pomocą indeksu i reprezentującego pozycję, którą musi przetworzyć Kernel z przestrzeni indeksów globalnych, jest wybrana tylko jedna cząstka.

Przykład operacji wektorowej. Odejmujemy dwa elementy, każdy złożony z wektora o wielkości 2. Nasze GPU może jednocześnie przetwarzać 4 elementy, więc idealny rozmiar wektora to 4, ale wymagałoby to znacznych zmian w programie i dodatkowej optymalizacji Kernela.

Aktualizacja pozycji cząstki w wyjściowym obiekcie off umieszczonym w pamięci. Ponownie, operacja wykonywana na danych typu wektorowego o rozmiarze 2, konwertowanych z float na int w sposób typowy dla ANSI C.

Rysunek 9. Przetwarzanie Kernela i zwracanie zmiennej *result.s*

deklarowaliśmy zmienne w Kernelu, musieliśmy podać kwalifikator określający, gdzie będą przechowywane argumenty. Wśród dostępnych kwalifikatorów przestrzeni pamięci dostępne są:

- **Global Memory** (pamięć globalna): dostępna w systemie dla Hosta i Urządzenia Przetwarzającego. Kwalifikator nosi nazwę ***__global***.
- **Constant Memory** (pamięć stałych): ma takie same cechy jak pamięć globalna, ale jest przeznaczona tylko do odczytu. Przed użyciem należy zainicjować wszystkie dane umieszczone w pamięci stałej. Kwalifikator: ***__constant***.
- **Local Memory** (pamięć lokalna): pamięć, która jest przydzielana dla przetwarzanych elementów. Jest ona specyfikowana dla grupy roboczej i dostępna tylko dla komponentów należących do danej grupy. Nie może być używana do przekazywania parametrów do Kernela. Kwalifikator: ***__local***.
- **Private Memory** (pamięć prywatna): dostępna dla pojedynczej instancji Kernela/elementu przetwarzanego, niewidoczna dla innych elementów roboczych. Pamięć domyślnie przeznaczona dla argumentów Kernela oraz zmiennych, jeśli nie zdecydowano inaczej. Kwalifikator: ***__private***.

Ważna cechą OpenCL jest wsparcie dla standardowych typów zmiennych numerycznych oraz niestandardowych, wektorowych, charakterystycznych dla operacji graficznych. Niektóre z typów zmiennych opisano niżej:

- **Skalarne typy danych –** typowe: *bool*, *char*, *int* i inne, znane z ANSI C. Warto zauważyć, że dostępny jest również typ *half* dla 16-bitowych liczb zmiennoprzecinkowych i *float* dla liczb 32-bitowych. Wsparcie dla 64-bitowych liczb zmiennoprzecinkowych *double* jest opcjonalne i definiowane za pomocą typu *cl_khr_fp64*.
- **Wektorowe typy danych:**
  - Bezpośrednio związane z tablicami ANSI C. **Nie wolno ani deklarować wektorów o dowolnych długościach, ani o zmiennej długości.**
  - Wspierane długości wektorów to: 2, 3, 4, 18 oraz 16. Dodatkowo trzeba uważać na ograniczenia sprzętowe związane z typem jednostki cieniującej, ponieważ ma to duży wpływ na wydajność.
  - Wektory są deklarowane za pomocą typów zmiennych skalarnych, po których bezpośrednio jest podawana długość wektora: *int**3**, int**16**, float**4**, double**16***.
  - Dostęp do elementów wektora może być realizowany na kilka sposobów, ale najczęściej używa się numeru pozycji wektora poprzedzanej przez token *s*. Na przykład, siódmy element w zmiennej *float16 my_vector* to *my_vector.s7*.

Każdy z wymienionych typów danych, skalarny lub wektorowy, może być używany jako element tablicy.

Uzbrojeni w nową wiedzę, przyjrzyjmy się implementacji Kernela i omówmy najważniejsze aspekty jego kodu źródłowego. Omówiono je w ramkach na **rysunkach 8** i **9**.

Podsumowanie

Mamy nadzieję, że artykuły będą pomocne w wykonaniu pierwszych kroków z OpenCL. Na pewno zaledwie pokazaliśmy wierzchołek góry lodowej rosnących wyzwań, poczynając od opracowania założeń aplikacji i kroków przetwarzania, jej implementacji oraz przede wszystkim – optymalizacji wydajności dla specyficznego Urządzenia Przetwarzającego. Jeśli zdecydujesz się Czytelniku zgłębiać tę domenę wiedzy, życzę Ci udanej, pouczającej podróży.

Penisoara Nicusor
Freescale Semiconductor