(General Purpose) GPU vs. CPU

- Graphics Processor Unit není jen na výpočet grafických efektů – lze ho využít i k urychlení matematických výpočtů – lze použít i third-party knihovny
- Pokud jsou data na sobě nezávislá, lze na jejich zpracování využít GPU
- V opačném případě a v případě kódu, který obsahuje množství podmínek, je lepší nechat výpočet na CPU
- Obecně platí, že GPU urychlí výpočet, může-li „tupě“ násobit a sčítat velké množství čísel, které na sobě nezávisí
    - Např. A=B+C a D=E+F
    - Ale už ne A=B+C; if A<C then D=E+F
    - CPU je totiž lepší v predikci skoků
    - Řešením může být kompletní přepsání algoritmu, aby vyhovoval GPU
        - Vždy to ale není možné

CUDA vs. OpenCL

- CUDA je proprietární technologie nVidie
- OpenCL je původně od Apple, ale dnes už otevřený standard
- CUDA je vizionář, což mu dalo náskok
- OpenCL zase běží i na hardware i ostatních výrobců
- OpenCL umí „fallback to CPU“, není-li k dispozici GPU
    - Jeden kód v programu, lépe se udržuje
        - Ale nejspíš bude pomalejší, než dobře napsaný kód pro CPU – tj. běží všude, ale někde pomalu

- V každém případě je ale třeba myslet jako GPU, ne jako CPU, jde-li nám o efektivní využití GPU

OpenCL

- Kódy příkladů, původní text a obrázky k OpenCL viz:
  - https://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201
  - http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/introductory-tutorial-to-opencl/
  - http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces

- Device – zařízení, kde se vykoná GPU-kód, tj. GPU, v případě fallbacku CPU
- Kernel – funkce v GPU-kódu, která se vykoná na Device
  - Aneb zkompilovaný program zapsaný v OpenCL
- Host – aka CPU
- Platform – Host + všechny dostupné Devices

- Součet vektorů na CPU

```
void vector_add_cpu (const float* src_a,
                const float* src_b,
                float*  res,
                const size_t num) {
   for (size_t i = 0; i < num; i++)
      res[i] = src_a[i] + src_b[i];
}
```

- Součet vektorů na GPU
  - kernel musí vracet void a být deklarován s __kernel

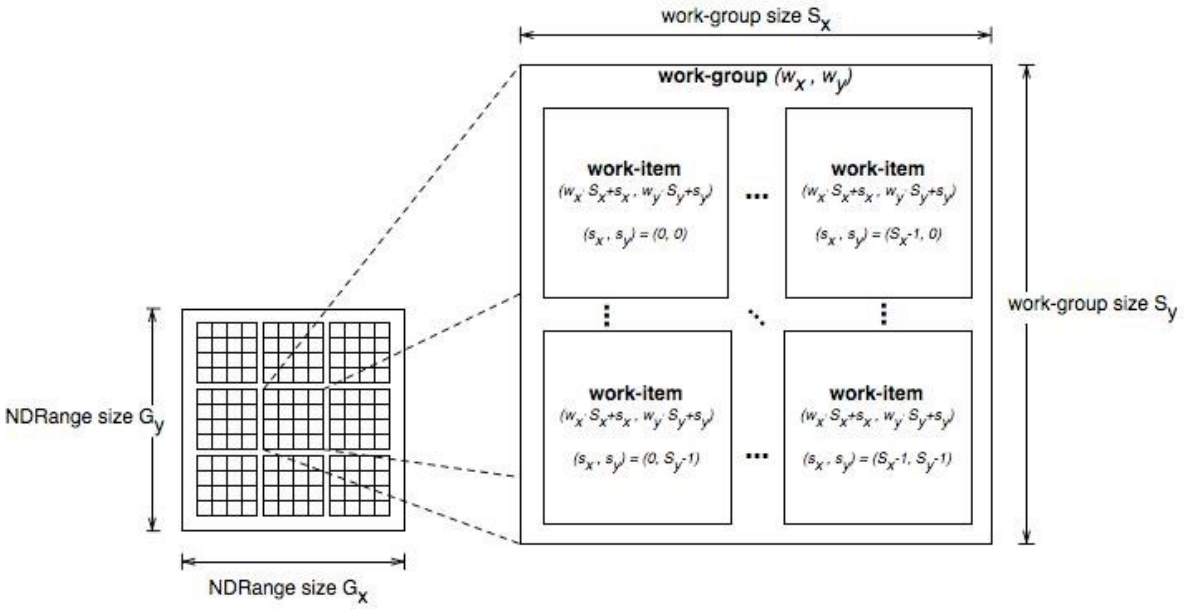
- pointer na paměť, se kterou se pracuje, musí být deklarován s __global

```
__kernel void vector_add_gpu (
            __global const float* src_a,
            __global const float* src_b,
            __global float* res,
           const int num) {
   /* get_global_id(0) vrátí ID aktuálního
      threadu. Jelikož jich na Device běží
      několik zároveň, každý sečte jenom svůj
      prvek vektoru. */

   const int idx = get_global_id(0);

   /* Podmínka je nutná, protože může být více
      threadů než prvků vektoru. Pokud by bylo
      více prvků vektoru než threadů, tak jeden
      thread může sečíst více prvků ala modulo.
   */
   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];
}
```

- prvek vektoru je tzv. work-item, které se sdružují do tzv. work-group, které se sdružují do tzv. ND-Range
- kernel se vykoná právě jednou pro každý work-item



- Ačkoliv lze mít jednu work-group, která obsáhne všechny work-items, není to dobrý nápad
  - Jedna workgroup se totiž celá počítá na jedné compute unit
    - Ale jedna compute unit může vykonat více work-groups
  - Takže v případě více compute units by ostatní compute units zůstaly nevyužité při jedné velké work-group, zatímco při několika work-groups se mohou využít všechny dostupné compute-units
  - Optimální počty závisí na použitím hardwaru, ale velikost work-group by měla být dělitelná 32 (nVidia warp) nebo 64 (AMD wavefront)
    - Je to vlastně to samé jako velikost vektoru SSE
  - Maximální velikost work-group je DEVICE_MAX_WORK_GROUP_SIZE,
    - případně KERNEL_WORK_GROUP_SIZE, kterou vrací fce GetKernelWorkGroupInfo, a která může být menší než DEVICE_MAX_WORK_GROUP_SIZE

- protože paměťové nároky kernelu mohou být různé v závislosti na jeho kódu, resp. na tom, kolik paměti/registrů použije na proměnné k ukládání mezivýsledků
  - což je omezeno použitým hw

- a zároveň musí platit, že velikost workgroup beze zbytku dělí počet work-items

- OpenCL program se překládá až na cílovém počítači přímo pro konkrétní hardware, který je tam k dispozici
- Z pohledu programu jde o tzv. kontext a frontu příkazů

```
cl_int error = 0;
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

// Platform
error = oclGetPlatformID(&platform);
if (error != CL_SUCCESS) {...}

// Device - CL_DEVICE_TYPE_GPU je typ zařízení,
//1 znamená, že chceme deskriptor 1x GPU
error = clGetDeviceIDs(platform,
        CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) {...}

// Context - zjistili jsme 1x GPU, tak vytvoříme
// kontext 1x GPU, ale lze dát i více devices
context = clCreateContext(0, 1, &device, NULL,
NULL, &error);
if (error != CL_SUCCESS) {...}

// Command-queue - fronta, do které budeme
//zadávat příkazy, co se má vypočítat
queue = clCreateCommandQueue(context, device, 0,
&error);
if (error != CL_SUCCESS) {...}
```

- Ale než něco spočítáme, je třeba alokovat paměť tak, aby ji Device viděl a mohl ji číst a zapisovat

```
float* src_a_h=new float[count];
cl_mem src_a_d = clCreateBuffer(context,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        mem_size, src_a_h, &error);
...
```

```
    //Žádný garbage collector, paměť se uvolňuje
    manuálně!

    delete[] src_a_h;
    clReleaseKernel(vector_add_k);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    clReleaseMemObject(src_a_d);
```

- Program -> kompilace -> Kernel -> výpočet
  - Pochopitelně stále platí, že co se dá udělat jednou, to se udělá jenom jednou a na začátku běhu programu, aby se to dalo opakovaně využívat a nezdržovalo to vlastní výpočet

```
//Kromě .c kódu taky existují C++ wrappery, se kterými
//může být život o něco jednodušší

//buď načteme .cl soubor, nebo to může být statický
//řetězec v kódu programu
ifstream file(kernelFile);
string prog(istreambuf_iterator<char>(file),
            (istreambuf_iterator<char>()));

cl::Program::Sources source( 1,
                             make_pair(prog.c_str(),
                             prog.length()+1));

cl::Program program(context, source);
file.close();

//zkompilujeme kernel pro dané zařízení
program.build(devices);

//a získáme kernel, kterému můžeme předat parametry a
//spustit ho
cl::Kernel kernel = cl::Kernel(program,
                               "kernel_function_name");
```

- Takže teď už v chybí jenom předat parametry a zařadit kernel do fronty ke spuštění

```
error = clSetKernelArg(vector_add_k, 0,
                       sizeof(cl_mem), &src_a_d);
error |= clSetKernelArg(vector_add_k, 1,
                        sizeof(cl_mem), &src_b_d);
error |= clSetKernelArg(vector_add_k, 2,
                        sizeof(cl_mem), &res_d);
error |= clSetKernelArg(vector_add_k, 3,
                        sizeof(size_t), &size);
assert(error == CL_SUCCESS);

// Launching kernel
const size_t local_ws = 512;  // Number of work-items
                              //per work-group
// shrRoundUp returns the smallest multiple of
// local_ws bigger than size
const size_t global_ws = shrRoundUp(local_ws, size);
    // Total number of work-items
```

  - global work-size je počet všech work-items
  - local size je počet work-items v jedné work-group

```
error = clEnqueueNDRangeKernel(queue, vector_add_k, 1,
          NULL, &global_ws, &local_ws, 0, NULL, NULL);
```

- A než se kernel dopočítá, můžeme mezitím dělat i jinou, užitečnou činnost
  - Poslední parametr je `clEnqueueNDRangeKernel` totiž event, který je signalizován v době dokončení kernelu
  - A podobně jsou předposlední dva parametry eventy, které musí být signalizovány, než se může daný kernel spustit

- Rekurze
  - Ne každý GPU hw to umí, a rozhodně to není dobrý nápad používat
  - GPU dosahuje ve výpočtech urychlení nad CPU právě proto, že GPU nemusí řešit stack a s ním související branch-prediction
    - A rozhodně není praktické kopírovat velké registry GPU na zásobník a zpět – push/pop

- typedef struct
  - lze použít vlastní datové typy
  - ale musí se kromě .c/.cpp kódu definovat i v .cl kódu

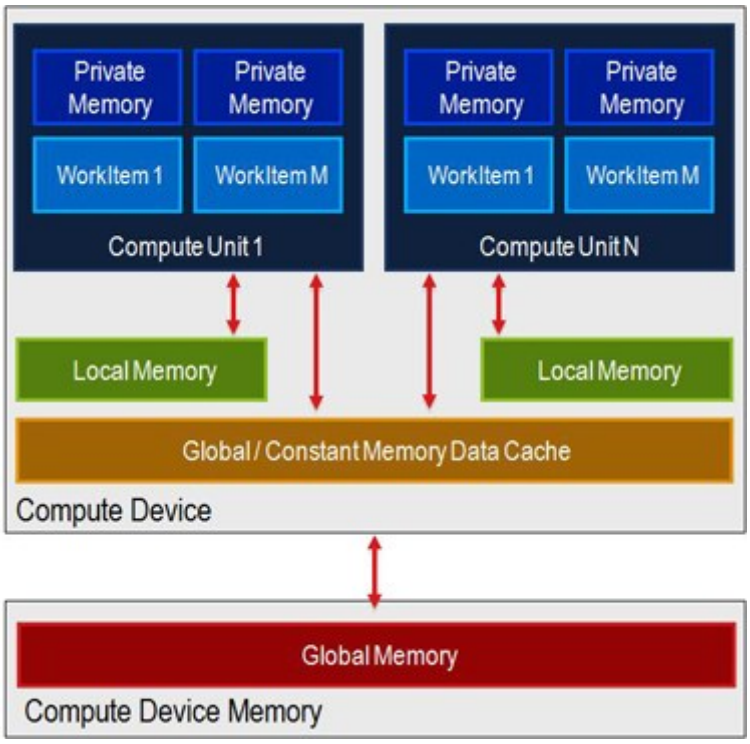
```
typedef char my_char[8];

typedef struct tag_my_struct
{
    long int        id;
    my_char         chars[2];
    int             numerics[4]
    float           decimals[4];
} my_struct;

__kernel void foo(__global my_struct * input,
                  __global int * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid].numerics[3]== 2 ? 1 : 0;
}
        http://stackoverflow.com/questions/2423825/cu
        stom-types-in-opencl-kernel
```

- Plánování vs. vykonání
  - work-items v jedné work-group budou naplánovány dohromady, ale už nikdo nezaručuje, že příslušné kernely budou vykonány zároveň
    - to závisí na ovladačích a hw, který se snaží zamaskovat zpoždění při práci s pamětí
      - nahrát data do paměti přístupné GPU a zpět něco stojí
      - takže na to pozor i při rozhodování, zda je daný problém vůbec vhodný pro výpočet na GPGPU

- Paměť



http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces

- Global
  - Nejpomalejší paměť celého GPU subsystému
  - Výkon závisí na tom, jak se s pamětí pracuje
  - Což závisí na konkrétním kusu hw a tudíž se vyplatí si pamatovat, že se má omezit konkurenční přístup ke stejné paměti z několik různých work-items, které jsou vykonávány (tj. mohly by být vykonávány) současně
    - Tzv. bank-conflict
    - Viz použití constant

- Private
  - Rychlá paměť pro použití při výpočtu jednoho work-item (tj. threadu GPU)
  - Její velikost, ani minimální ani maximální, není definována žádným standardem, takže závisí na konkrétním hw
  - Takže z toho vyplývá jedno pravidlo:
    - Použít jí co nejméně
    - Protože jakmile nároky na privátní paměť překročí limit hw, tak se to uloží do globální paměti a to degraduje výkon
- Local
  - Opět je rychlejší než globální paměť
  - Slouží ke sdílení dat mezi jednotlivými work-items v jedné work-group

- Constant
  - Read-only paměť
  - Dává hw možnost, aby optimalizoval konkurenční přístup k paměti, která je jenom ke čtení
    - Tj. aby se vyhnul onomu bank-conflict

- Násobení vektorů
  - Viz
    - http://developer.amd.com/community/blog/efficient-dot-product-implementation-using-persistent-threads/
  - Nejprve by se spustil kernel, který vynásobí prvky vektorů

```
__kernel void dot_mul_kernel(
          __global const double * x, // input vector
          __global const double * y, // input vector
          __global double * r, // result vector
          uint n) { // input vector size

      uint id = get_global_id(0);
      if ( id < n ) {
         r[id] = x[id] * y[id];
              // multiply elements, store product
      }
}
```

  - Aby se pak pustil další kernel, který je sečte
  - Anebo se to dá udělat ještě takhle

```
#define LOCAL_GROUP_XDIM 256
__kernel __attribute__(
       (reqd_work_group_size(LOCAL_GROUP_XDIM, 1, 1)))
    void dot_local_reduce_kernel(
            __global const double * x, // input vector
            __global const double * y, // input vector
            __global double * r, // result vector
            uint n) { // input vector size
    uint id = get_global_id(0);
    uint lcl_id = get_local_id(0);
    uint grp_id = get_group_id(0);
    double priv_acc = 0;
                      // accumulator in private memory
    __local double lcl_acc[LOCAL_GROUP_XDIM];
                      // accumulators in local memory
```

```
if ( id < n ) {
  priv_acc = lcl_acc[lcl_id] = x[id] * y[id];
             // multiply elements, store product
}

barrier(CLK_LOCAL_MEM_FENCE);
     // Find the sum of the accumulators.

uint dist = LOCAL_GROUP_XDIM;
     // i.e., get_local_size(0);

while ( dist > 1 ) {
   dist >>= 1;
   if ( lcl_id < dist ){
     // Private memory accumulator
     //avoids extra local memory read.

     priv_acc += lcl_acc[lcl_id + dist];
     lcl_acc[lcl_id] = priv_acc;
   }

    barrier(CLK_LOCAL_MEM_FENCE);
}

//Store the result (the sum for the local work group).

if ( lcl_id == 0 ) {
    r[grp_id] = priv_acc;
}

} //konec kernelu
```

- Barrier vs. Fence
  - Bariéra slouží k synchronizaci vláken vykonávajících kernel na jednotlivých work-items v jedné work-group
    - Různé work-groups mezi sebou nelze synchronizovat
  - Fence zajistí, že se všechny load/store instrukce dokončí před fence, tj. předtím, než se začnou vykonávat další load/store instrukce, které jsou v programovém kódu po fence
  - Fence se vztahuje jenom na jeden work-item
  - Bariéra může volat fence pro všechny work-items

CUDA

- Idea je v podstatě stejná
- Pokud jste pochopili OpenCL, CUDA by neměla být problém po adaptaci na její terminologii
  - Viz http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl/

Coalesced Memory

- Přístup k paměti, jehož nedodržení může být za následek výrazné zpomalení výpočtu
  - Až si možná budete říkat, že daný problém není pro GPGPU vhodný
    - Nicméně pokud někdo úspěšně řeší stejný problém, pak je chyba na vaší straně – a právě, že dost možná zde

- Každý thread má svoje ID, takže lze v paměti vzestupně uspořádat vlákna i prvky, ke kterým vlákna, přistupují
  - Takový přístup je coalesced

- Příkladem, máme v paměti matici 3x3 a chceme najít nejmenší prvek

  0x100: 0 1 2
  0x118: 3 4 5
  0x130: 6 7 8

- Každý thread si vezme jeden řádek v něm najde své lokální minimum

  Thread 0: 0 1 2
  Thread 1: 3 4 5
  Thread 2: 6 7 8

- Protože prvky matice jsou v paměti vzestupně uložené jakoby v jedné řádce, je to coalesced memory access
  - Offset = row*rowsize+column

- Ale kdyby byla uložena v paměti po sloupcích

  0x100: 0 3 6
  0x118: 1 4 7
  0x130: 2 5 8

- Tak by to vlákna četla z paměti ne sekvenčně, jak to hw vyhovuje, ale napřeskáčku, což je přístup, na který nebyl hw navržený, a proto se můžete dočkat zpomalení

- Jak tedy alokovat double **matrix ?

```
matrix = (double**) malloc(sizeof(double)*rows*cols+
                           sizeof(double*)*rows);

for (size_t i=0; i<rows; i++) {
  matrix[i] = (double*) (
                (unsigned char*) (&matrix)+
                sizeof(double*)*rows+
                sizeof(double)*i
              );
}

matrix[row][col] = ….
```

- Případně se dá ještě na konkrétním systému použít taková alokační funkce, která zajistí, že **matrix bude alokovaná na optimálně zarovnané adrese
  - posix_memalign
  - _aligned_malloc //Windows
  - Nebo si to programátor může zarovnat sám přičtením příslušného offsetu

- Coalesced memory je dobrým příkladem toho, že
  - jedna věc je znát optimální algoritmus k řešení dané úlohy
  - a druhá věc je tento algoritmus umět implementovat tak, abychom se předpokládaného urychlení vůbec dočkali!