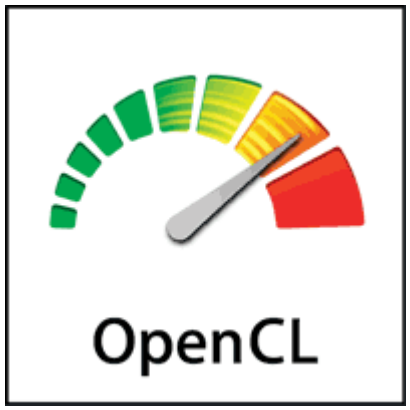


Laboratory of Information Technologies

OpenCL

Computation on HybriLIT

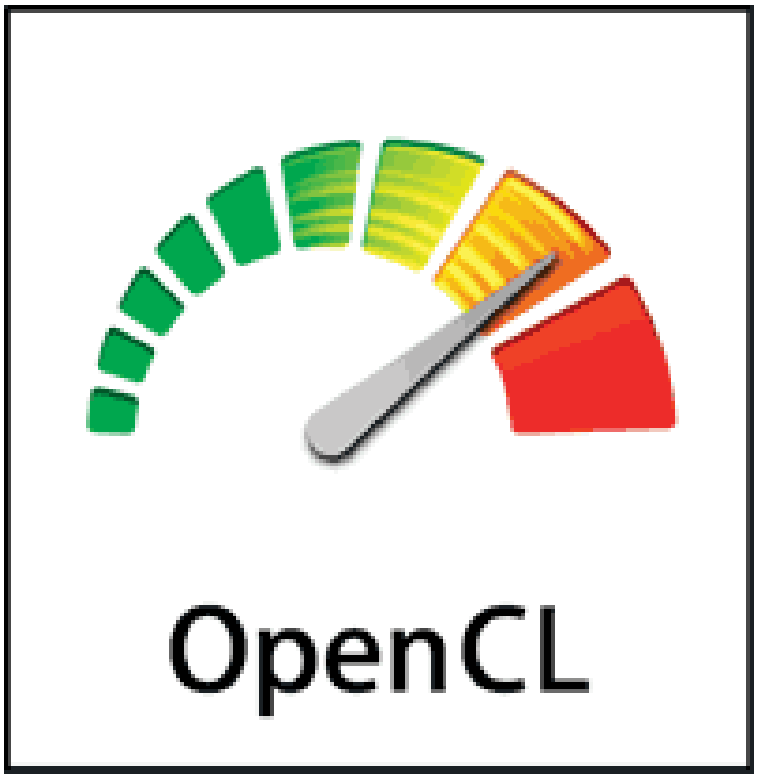
Brief introduction and getting started

Alexander Ayriyan
Laboratory of Information Technologies
Joint Institute for Nuclear Research

05.09.2014 (Friday)
Tutorial in frame of Helmholtz International Summer School'2014
Lattice QCD, Hadron Structure andHadronic Matter
JINR, Dubna, 25.08-06.09.2014

What OpenCL is

Open Computing Language (OpenCL) is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL is based on C99.



Credit: https://www.khronos.org/opencl/

OpenCL as a standard

Credit: https://www.khronos.org/

How OpenCL works

OpenCL consists of a kernel programming API used to program a co-processor or GPU and a run-time host API used to coordinate the execution of these kernels and perform other operations such as memory synchronization. The OpenCL programming model is based on the parallel execution of a kernel over many threads to exploit SIMD or/and SIMT architectures.

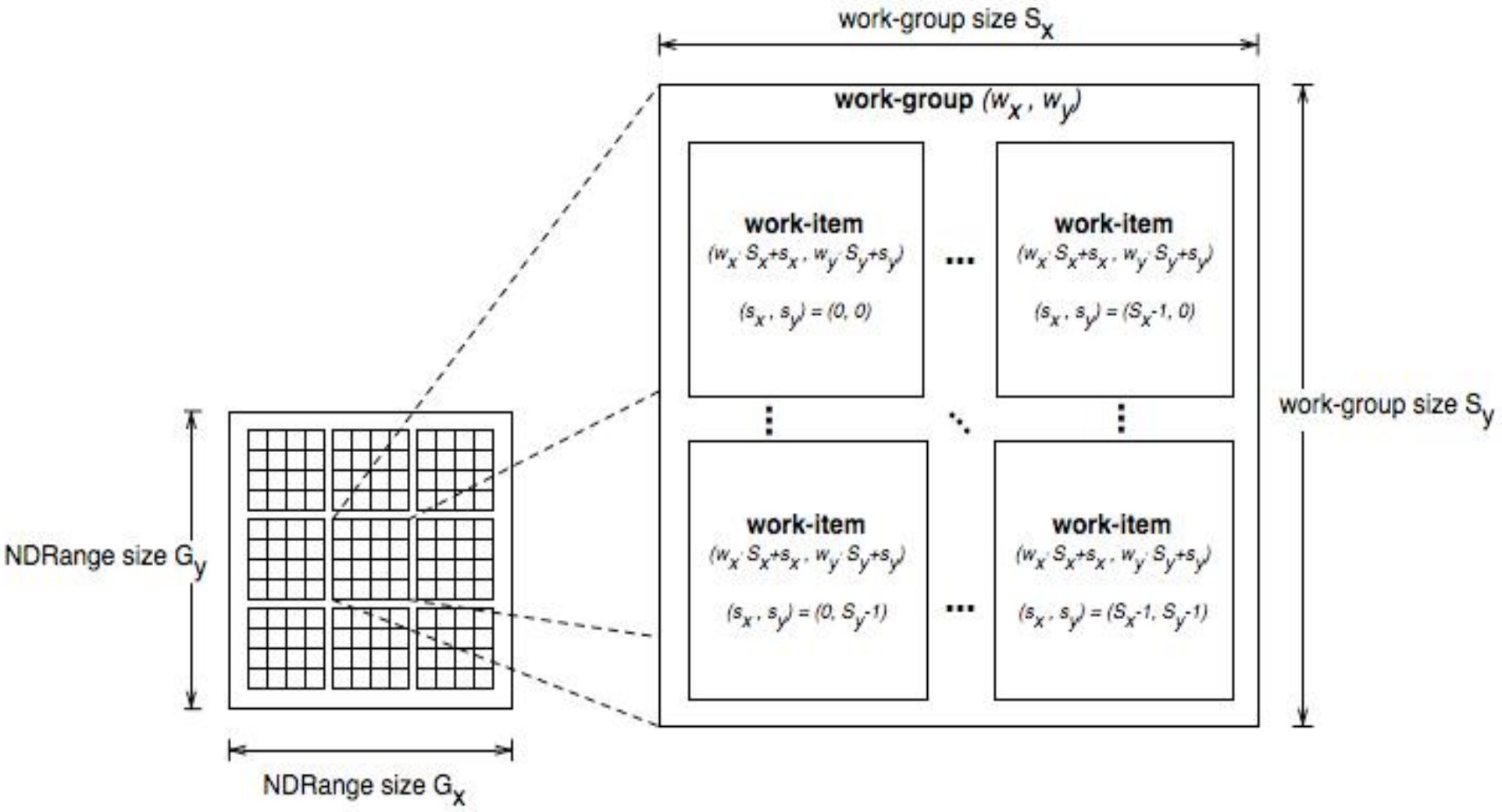
SIMD - **S**ingle **I**nstruction, **M**ultiple **D**ata
SIMT - **S**ingle **I**nstruction, **M**ultiple **T**hreads

Single Instruction is realized in a kernel.
Multiple Data is hosted in data massives.
Multiple Threads parallely execute a kernel.

The **same code** is executed in parallel mode by a **different thread**, and each thread executes the code with **different data**.

http://www.parallella.org/

OpenCL Programming model (1)



Credit: http://opencl.codeplex.com/

OpenCL Programming model (2)

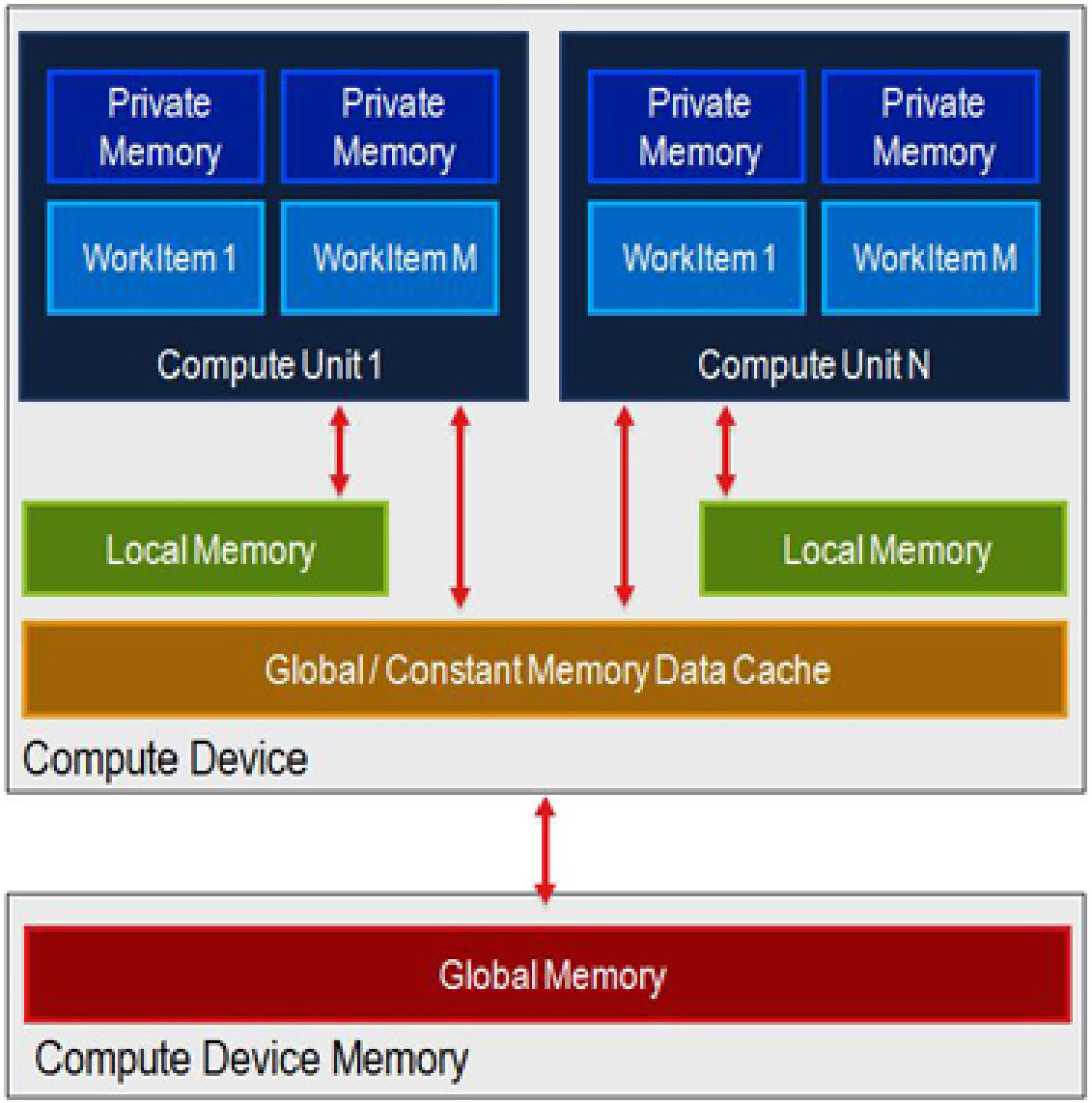
The **work-items** are the smallest execution entity (equivalent to the CUDA threads). Every time a kernel is launched, lots of work-items (a number specified by the programmer) are launched executing the same code. Each work-item has an **ID**, which is accessible from a kernel, and which is used to distinguish the data to be processed by each work-item.

The **work-groups** exist to allow communication and cooperation between work-items (equivalent to CUDA thread blocks). They reflect how work-items are organized (it's a N-dimensional grid of work-groups, with N = 1, 2 or 3). Work-groups also have an unique ID that can be refereed from a kernel.

The ND-Range is the next organization level, specifying how work-groups are organized (again, as a N-dimensional grid of work-groups, N = 1, 2 or 3);

http://opencl.codeplex.com/

OpenCL Memory Hierarchy (1)



OpenCL	CUDA
Global	Global
Local	Shared
Private	Register

Rob Farber, http://www.codeproject.com

OpenCL Memory Hierarchy (2)

Following are the OpenCL address qualifiers (which are distinct from access qualifiers):

- **`__global`**: memory allocated from global address space
- **`__constant`**: a region of read-only memory
- **`__local`**: memory shared by work-group
- **`__private`**: private per work-item memory

Note: kernel arguments have to be **`__global`**, **`__constant`** or **`__local`**. Pointers that are cast using different address qualifiers are undefined.

OpenCL Program Structure (1)

1. Get a platform information

```
clGetPlatformIDs(1, &platform_id, &num_platforms);
```

2. Get information about devices

```
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
&num_devices);
```

3. Create an OpenCL context

```
context = clCreateContext(NULL, CL_DEVICE_TYPE_GPU, &device_id,
                          NULL, NULL, NULL);
```

4. Create a command queue

```
queue = clCreateCommandQueue(context, device_id, 0, NULL);
```

OpenCL Program Structure (2)

5. Create a program from the kernel source and build it

```
program = clCreateProgramWithSource(context, 1,
    (const char**)&source_str, (const size_t*)&source_size, NULL);
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

where `source_str` consists of kernel program (kernel source)

```
char *source_str = {
"__kernel void hello_world(__global int *A) { \n"
"// Get the index the current work-item       \n"
"int i = get_global_id(0);                     \n"
"// Write this number in data massive          \n"
"A[i] = i;                                     \n"
"}                                             \n"};
```

```
// The kernel file "helloWorld_kernel.cl"
__kernel void hello_world(__global int *A)
{
    // Get the index the current work-item
    int i = get_global_id(0);
    // Write this number in data massive
    A[i] = i;
}
```

OpenCL Program Structure (3)

6. Create an OpenCL kernel

```
kernel = clCreateKernel(program, "hello_world", NULL);
```

7. Create device memory buffer and copy the vector A from the host

```
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_COPY_HOST_PTR,
                                  nVector*sizeof(int), A, NULL);
```

8. Set the kernel arguments

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&a_mem_obj);
```

9. Execute the OpenCL kernel

```
size_t global_item_size = nVector; // Number of work-items (threads)
size_t local_item_size = 64; // Size of work-groups
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_item_size,
                       &local_item_size, 0, NULL, NULL);
```

10. Get back data, copy buffer `a_mem_obj` to the local variable `A`

```
clEnqueueReadBuffer(queue, a_mem_obj, CL_TRUE, 0,
                    nVector*sizeof(int), A, 0, NULL, NULL);
```

How to Get Device Properties (1)

```
//   Get list of available platforms
     cl_platform_id platform_id = NULL;
     clGetPlatformIDs(1, &platform_id, NULL);

//   Get information about available platforms
     clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, MAX_NAME_SIZE, name, NULL);
     clGetPlatformInfo(platform_id, CL_PLATFORM_VERSION, MAX_NAME_SIZE, name, NULL);

//   Get list of available devices on platforms
     clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ALL, MAX_NUM_DEVICES, devices, &nDevices);

//   Get information about devices
     for(int iDevices = 0; iDevices < nDevices; ++iDevices)
     {
          clGetDeviceInfo(devices[iDevices], CL_DEVICE_NAME, MAX_NAME_SIZE, name, NULL);
          clGetDeviceInfo(devices[iDevices], CL_DRIVER_VERSION, MAX_NAME_SIZE, name, NULL);
          clGetDeviceInfo(devices[iDevices], CL_DEVICE_GLOBAL_MEM_SIZE,
                                                  sizeof(cl_ulong),&entries, NULL);
          clGetDeviceInfo(devices[iDevices], CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,
                                                  sizeof(cl_ulong),&entries, NULL);
          clGetDeviceInfo(devices[iDevices], CL_DEVICE_MAX_CLOCK_FREQUENCY,
                                                  sizeof(cl_ulong),&entries, NULL);
          clGetDeviceInfo(devices[iDevices], CL_DEVICE_MAX_WORK_GROUP_SIZE,
                                                  sizeof(size_t),&p_size, NULL);
     }
```

How to Get Device Properties (2)

```
-bash-4.1:~/OpenCL/Tutorial/FindDevice$ nvcc findDevice.c -o findDevice -l OpenCL
-bash-4.1:~/OpenCL/Tutorial/FindDevice$ srun findDevice
CL_PLATFORM_NAME = NVIDIA CUDA
CL_PLATFORM_VERSION = OpenCL 1.1 CUDA 4.2.1
Found 1 devices
Device name = GeForce GT 635M
    Driver version = 319.60
    Global Memory (MB):     2047
    Local Memory (KB):      32
    Max clock (MHz) :       950
    Max Work Group Size:    1024
-bash-4.1:~/OpenCL/Tutorial/FindDevice$
```

Notebook ASUS: Intel Core $i7$, nVidia GeForce GT 635M (2GB)

“Hello World” (1)

```
//  Get platform and device information
  clGetPlatformIDs(1,&platform_id,&num_platforms);
  clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices);
//  Create an OpenCL context
  cl_context context=clCreateContext(NULL,CL_DEVICE_TYPE_GPU,&device_id,NULL,NULL,NULL);
//  Create a command queue
  cl_cmd_queue cmd_queue = clCreateCommandQueue(context,device_id,0,NULL);
//  Create a program from the kernel source
  cl_program program=clCreateProgramWithSource(context,1,(const char**)&source,NULL,NULL);
//  Build the program
  clBuildProgram(program,1,&device_id,NULL,NULL,NULL);
//  Create the OpenCL kernel
  cl_kernel kernel=clCreateKernel(program,"hello_world",NULL);
//  Create memory buffer on the device for vector
  cl_mem a_mem_obj=clCreateBuffer(context,CL_MEM_COPY_HOST_PTR,nVector*sizeof(int),A,NULL);
//  Set the kernel arguments
  clSetKernelArg(kernel,0,sizeof(cl_mem),(void*)&a_mem_obj);
//  Execute the OpenCL kernel
  size_t global_size = nVector; size_t local_size = 64;
  clEnqueueNDRangeKernel(cmd_queue,kernel,1,NULL,&global_size,&local_size,0,NULL,NULL);
//  Copy the memory buffer a_mem_obj on the device to the local variable A
  clEnqueueReadBuffer(cmd_queue,a_mem_obj,CL_TRUE,0,nVector*sizeof(int), A ,0,NULL,NULL);
//  Print the result to the screen
  for(int iVector = 0; iVector < nVector; iVector++)
    printf("Helo World! I am thread number %d.\n", A[iVector]);
```

“Hello World” (2)

```
// The kernel file “helloWorld_kernel.cl”
__kernel void hello_world(__global int *A)
{
    // Get the index the current work-item
    int i = get_global_id(0);
    // Write this number in data massive
    A[i] = i;
}
```

```
-bash-4.1:~/OpenCL/Tutorial/HelloWorld$ nvcc helloWorld.c -o helloWorld -l OpenCL
-bash-4.1:~/OpenCL/Tutorial/HelloWorld$ srun helloWorld
Helo World! I am thread number 0.
Helo World! I am thread number 1.
Helo World! I am thread number 2.
...
Helo World! I am thread number 1021.
Helo World! I am thread number 1022.
Helo World! I am thread number 1023.
-bash-4.1:~/OpenCL/Tutorial/HelloWorld$
```

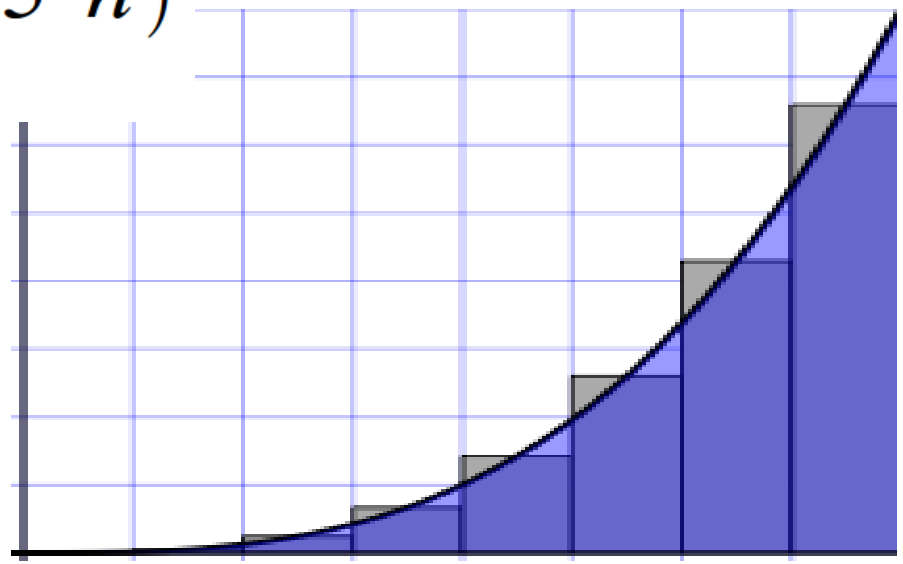
Calculation of Integral (1)

Calculate numerically integral of function $\mathbf{f(x)}$ on the interval $[\mathbf{a,b}]$.

$$I=\int_a^b f(x)\,dx=\int_0^{\frac{\pi}{2}} \sqrt{x}\cdot\left[\sin(x)\cdot\ln(x+1)\right]^{\frac{x}{5}}\,dx$$

The answer is $\mathbf{I\approx 1.18832595716124}$

$$\int_a^b f(x)\,dx\approx h\sum_{i=0}^{i=n-1} f(x_i+0.5\cdot h)$$



The approximation error formula

$$R(x)=\frac{f''(x)}{24}(b-a)h^2$$

Calculation of Integral (2)

```
double func(double x) { return pow( sin(x) * log(x+1.0), x/5.0 ) * pow(x, 0.5); }
int main()
{
    double h = (right - left) / (N - 1);
    for(int i = 0; i < N; i++)
        X[i] = left + i * h;

    double integral = 0.0;
    for(int i = 0; i < N-1; i++)
    {
        Y[i] = h * func(0.5 * (X[i] + X[i+1]));
        integral += Y[i];
    }
    printf("Integral = %e\n", integral);
    return 0;
}
```

```
-bash-4.1:~/OpenCL/Tutorial/HelloWorld$ c++ calcInteg.cpp -o calcInteg -O3
-bash-4.1:~/OpenCL/Tutorial/CalcIntegral$ srun calcInteg
Calculation of the integral on CPU!
Exact answer is 1.18832595716124
Integral = 1.18832595716153
Time = 13.77 sec.
-bash-4.1:~/OpenCL/Tutorial/CalcIntegral$
```

Calculation of Integral (3)

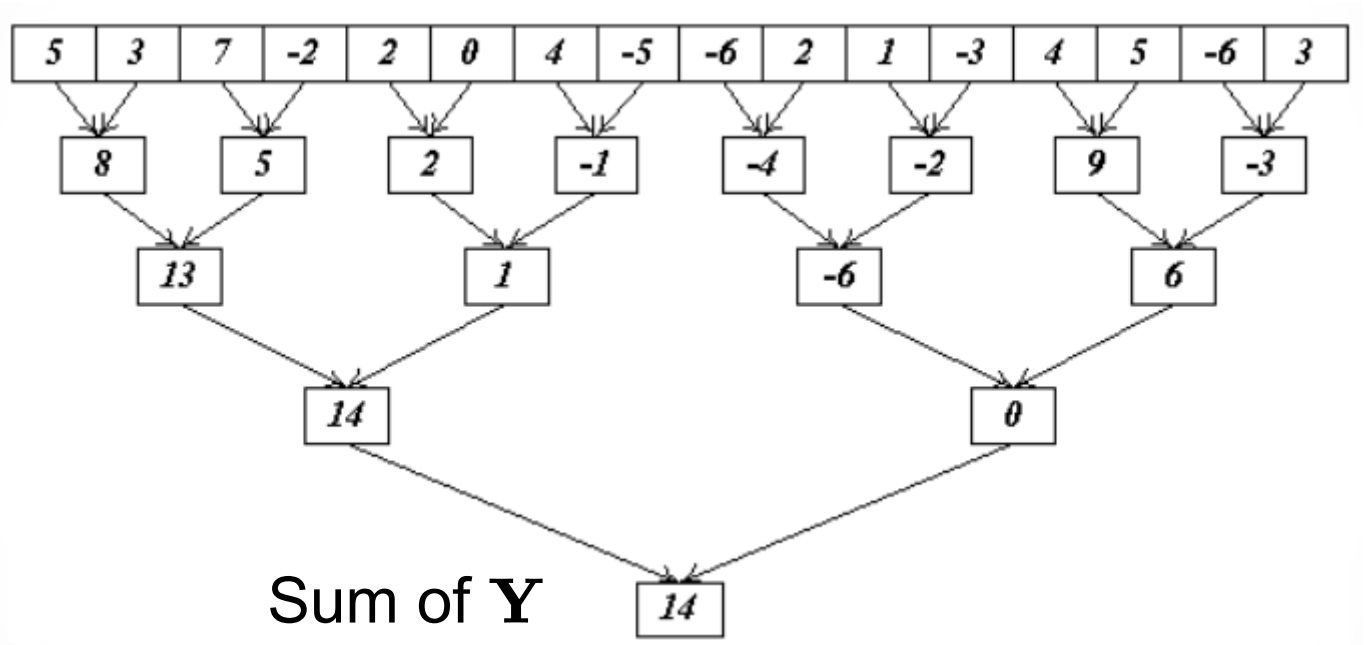
```
#pragma OPENCL EXTENSION cl_khr_fp64: enable
double func(double x) { return pow( sin(x) * log(x+1.0), x/5.0 ) * pow(x, 0.5); }

__kernel void calc_integral(__global double *h, __global double *X, __global double *Y)
{
//   Get the index of the current element
     int i = get_global_id(0);
//   Do the operation
     Y[i] = h[0] * func(0.5 * (X[i] + X[i+1]));
}
```

```
-bash-4.1:~/OpenCL/Tutorial/HelloWorld$ nvcc calcInteg.c -o calcInteg -l OpenCL
-bash-4.1:~/OpenCL/Tutorial/CalcIntegral$ srun calcInteg
Calculation of the integral on GPU!
Exact answer is 1.18832595716124
Integral = 1.18832595716153
Time = 2.88 sec.
-bash-4.1:~/OpenCL/Tutorial/CalcIntegral$
```

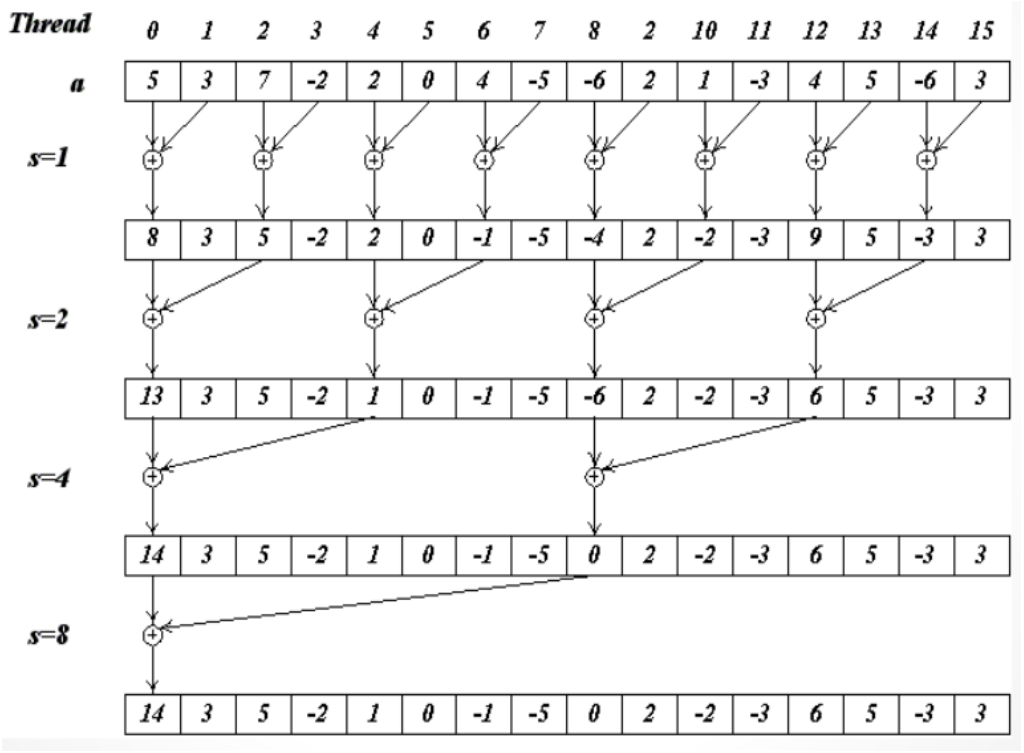
Calculation on GPU is around 5 times faster then on CPU.

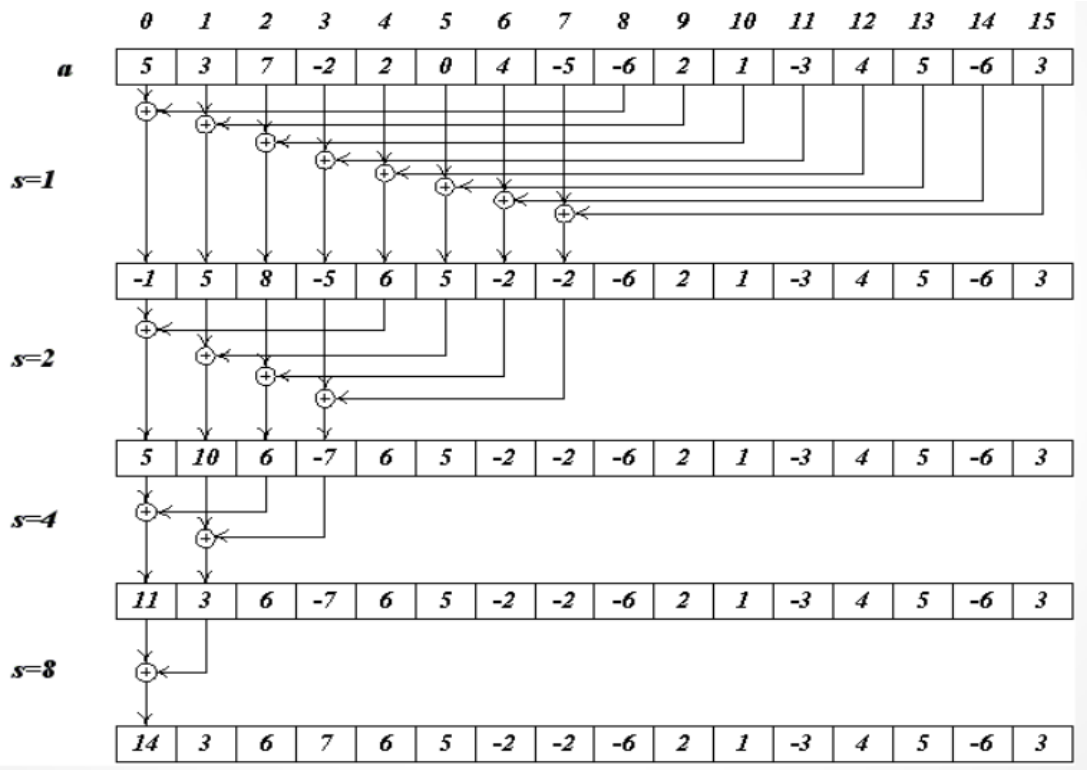
Parallel Reduction Algorithms (1)

A vector $\mathbf{Y}$



Parallel Reduction Algorithms (1)





J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*

Reduction for Sum Operator (1)

```
__kernel void calc_parallel_sum(__global int *gData, __local int *lData)
{
    int iGlobal = get_global_id(0);
    int iLocal  = get_local_id(0);
    int sizeGroup = get_local_size(0);
    int iGroup = get_group_id(0);

    lData[iLocal] = gData[iGlobal];  // load into shared memory
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=1; s <sizeGroup; s *= 2)
    {
        if(iLocal % (2*s) == 0)
            lData[iLocal] += lData[iLocal + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if(iLocal == 0)  // write result of block reduction
        gData[iGroup] = lData[0];
}
```

J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*

Reduction for Sum Operator (2)

```
__kernel void calc_parallel_sum(__global int *gData, __local int *lData)
{
    int iGlobal = get_global_id(0);
    int iLocal  = get_local_id(0);
    int sizeGroup = get_local_size(0);
    int iGroup = get_group_id(0);

    lData[iLocal] = gData[iGlobal];  // load into shared memory
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=1; s <sizeGroup; s *= 2)
    {
        int index = 2 * s * iLocal;
        if(index < sizeGroup)
            lData[iLocal] += lData[iLocal + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if(iLocal == 0)  // write result of block reduction
        gData[iGroup] = lData[0];
}
```

J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*

Reduction for Sum Operator (3)

```
__kernel void calc_parallel_sum(__global int *gData, __local int *lData)
{
    int iGlobal = get_global_id(0);
    int iLocal  = get_local_id(0);
    int sizeGroup = get_local_size(0);
    int iGroup = get_group_id(0);

    lData[iLocal] = gData[iGlobal];  // load into shared memory
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s = sizeGroup/2; s > 0; s >>= 1)
    {
        if(iLocal < s)
            lData[iLocal] += lData[iLocal + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if(iLocal == 0)  // write result of block reduction
        gData[iGroup] = lData[0];
}
```

J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*