UNIVERSIDAD DEL BÍO-BÍO
FACULTAD DE CIENCIAS EMPRESARIALES

Programming Frameworks
Heterogeneous Computing

Professor: Dr. Joel Fuentes - jfuentes@ubiobio.cl

Teaching Assistants:
- Daniel López - daniel.lopez1701@alumnos.ubiobio.cl
- Sebastián González - sebastian.gonzalez1801@alumnos.ubiobio.cl

Course website: http://www.face.ubiobio.cl/~jfuentes/classes/ch

Contents

- CUDA
- SYCL (DPC++)

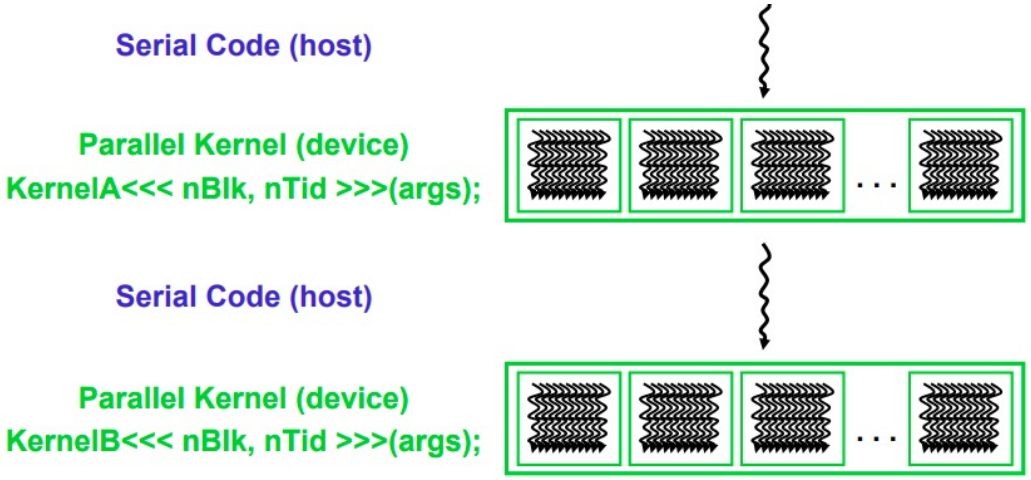
Programming Languages and Frameworks

- CPU multi-core
  - C++, Java, OpenMP, DPC++
- GPU
  - OpenCL, CUDA, DPC++, OpenACC
- FPGA
  - OpenCL, DPC++

CUDA

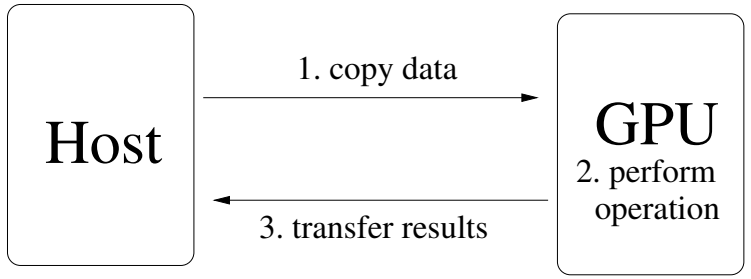
CUDA

- Compute Unified Device Architecture
- Extended C programming
- Serial code programming on Host (CPU)
- Parallel code programming on Device (GPU)



CUDA

- GPU can run many threads simultaneously, but not independently
- Threads on Device connected in groups called warps
- All members of a warp execute the same operation
  - SIMT = Single Instruction, Multiple Threads
- Programmer writes function that runs on the device (kernel)
- Function is invoked with a number of blocks
- All threads execute the same function
- Host and GPU have separate memory spaces
- Memory must be explicitly transferred



CUDA

- Extended C

Declarations
global, device, shared, local, constant

Keywords
threadIdx, blockIdx

Intrinsics
__syncthreads

Runtime API
- Memory, symbol, execution management

```
#include <stdio.h>

__global__ void hello() {
     int id = threadIdx.x + blockIdx.x * blockDim.x;
     printf("Hello from thread %d (%d of block %d)\n",
     id, threadIdx.x, blockIdx.x);
}

int main() {
     //launch 3 blocks of 4 threads each
     hello<<<3,4>>>();

     //make sure kernel completes
     cudaDeviceSynchronize();
}
```

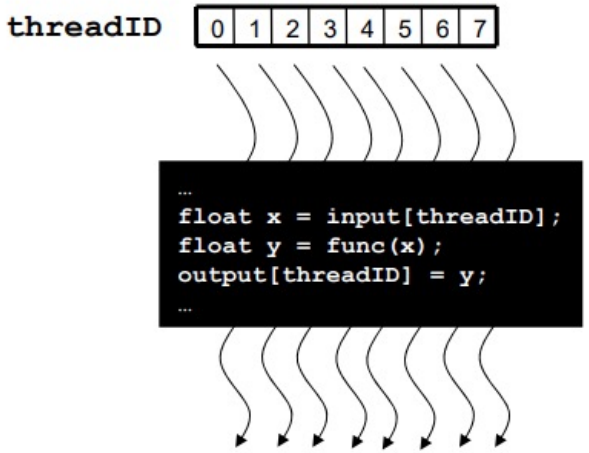
CUDA

- Possible output of the example
- Threads and blocks are executed in any order

```
Hello from thread 0 (0 of block 0)
Hello from thread 1 (1 of block 0)
Hello from thread 2 (2 of block 0)
Hello from thread 3 (3 of block 0)
Hello from thread 8 (0 of block 2)
Hello from thread 9 (1 of block 2)
Hello from thread 10 (2 of block 2)
Hello from thread 11 (3 of block 2)
Hello from thread 4 (0 of block 1)
Hello from thread 5 (1 of block 1)
Hello from thread 6 (2 of block 1)
Hello from thread 7 (3 of block 1)
```

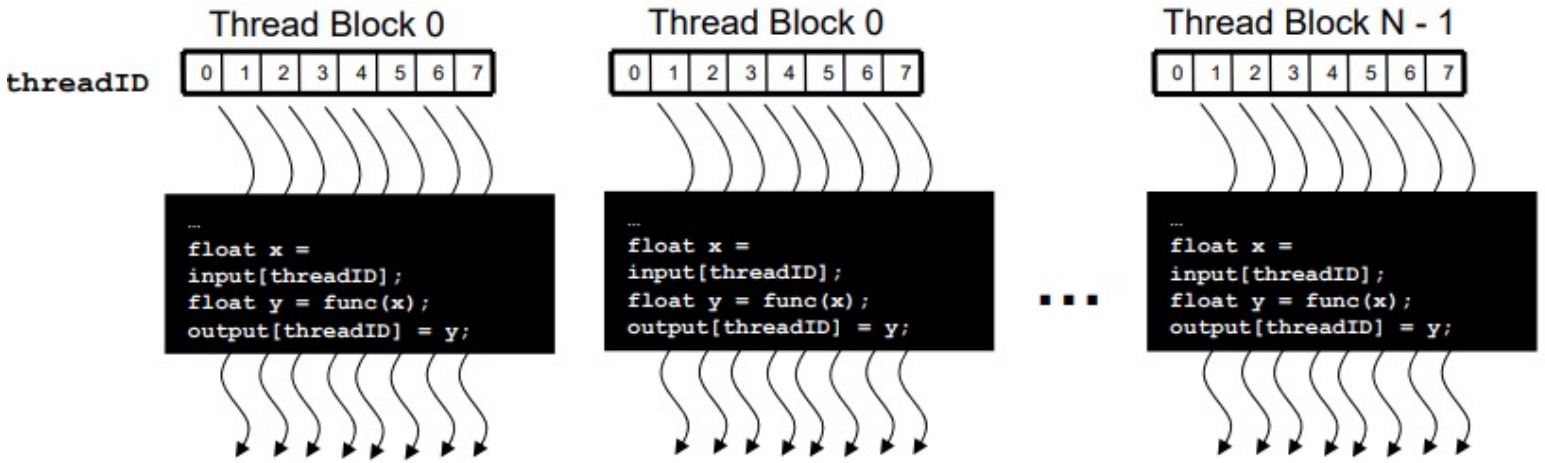
CUDA

- In CUDA all threads run the same code
- Each thread has its ID which is used to calculate memory accesses and make control decisions



CUDA

- Threads are organized into multiple blocks
- Threads in a block can cooperate through the use of shared memory, atomic operations, and synchronization barriers.
- Threads in different blocks cannot cooperate.



CUDA: Ejemplo suma de vectores

```
int main() {
    int* a;          //first input array (on host)
    int* a_dev;      //first input array (on device)

    a = (int*) malloc(N*sizeof(int));
    cudaMalloc((void**) &a_dev, N*sizeof(int));

    …                //same for b and res

    free(a);
    cudaFree(a_dev);
}
```

- On the host:
  1. **Reserve memory on device**
  2. Copy data to device
  3. Call the kernel
  4. Copy results to the host
  5. **Free device memory**
- On device:
  1. __global__
  2. Determine thread ID

CUDA: Ejemplo suma de vectores

```
int main() {
    …
    int threads = 512;         //# threads per block
    int blocks  = (N+threads-1)/threads;
                               //# blocks (N/threads rounded up)

    kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
    …
}
```

- En el host:
  1. Reserve memory on device
  2. Copy data to device
  3. **Call the kernel**
  4. Copy results to the host
  5. Free device memory
- On device:
  1. __global__
  2. Determine thread ID

CUDA: Ejemplo suma de vectores

```
__global__ void kernel(int* res, int* a, int* b) {
      //sets res[i] = a[i] + b[i]
      //each thread is responsible for one value of i

    int thread_id = threadIdx.x + blockIdx.x*blockDim.x;

    if(thread_id < N) {
         res[thread_id] = a[thread_id] + b[thread_id];
    }
}
```

- En el host:
  1. Reserve memory on device
  2. Copy data to device
  3. Call the kernel
  4. Copy results to the host
  5. Free device memory
- **On device**:
  - __global__
  - Determine thread ID

SYCL

SYCL

- SYCL is a proposal for XPU (GPU, CPU and FPGA).
- It corresponds to a C++ extension to support heterogeneous programming.
- Built from OpenCL fundamentals (evolution)
- Developed by the Khronos https://www.khronos.org/ group

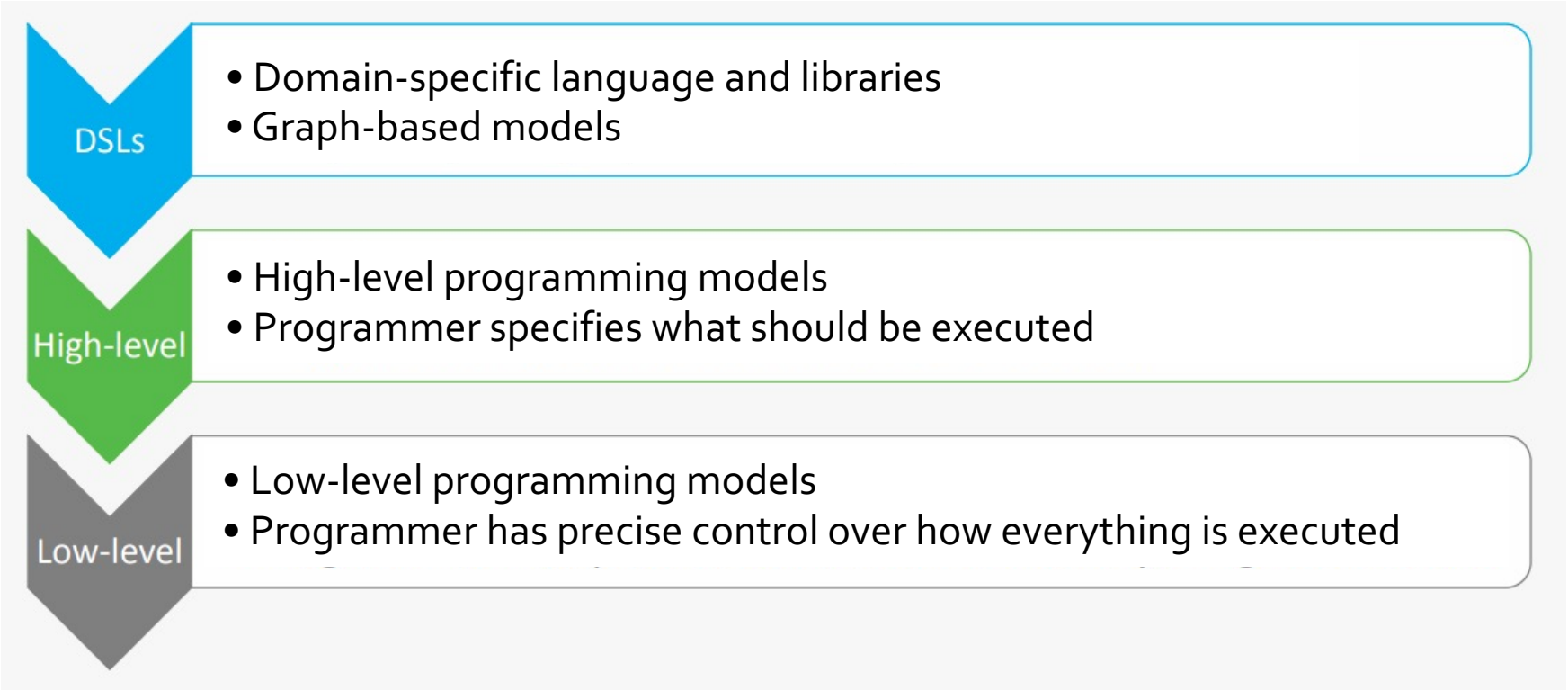
SYCL

- Platforms that support SYCL:

Implementation	Supported Platforms	Supported Devices	Required Version
ComputeCpp	Windows 10 Visual Studio 2019 (64bit)* Ubtuntu 18.04 (64bit)	Intel CPU (OpenCL) Intel GPU (OpenCL)	CE 2.4.0
DPC++	Intel DevCloud Windows 10 Visual Studio 2019 (64bit) Red Hat Enterprise Linux 8, CentOS 8 Ubtuntu 18.04 LTS, 20.04 LTS (64bit) Refer to System Requirements for more details	Intel CPU (OpenCL) Intel GPU (OpenCL) Intel FPGA (OpenCL) Nvidia GPU (CUDA)**	2021.2
hipSYCL	Any Linux	CPU (OpenMP) AMD GPU (ROCm)*** Nvidia GPU (CUDA)	Latest develop branch

SYCL

- Heterogeneous languages



Source: Alastair Murray. Codeplay. Layering Abstractions, Heterogeneous Programming and Performance Portability. 2017.

SYCL

- Problem: Different hardware requires different programming approaches
- Functionality can be portable, but not performance
- An algorithm optimized for a particular architecture can run very badly on another.

Partial solution: Consider a higher-level language.

Source: Alastair Murray. Codeplay. Layering Abstractions, Heterogeneous Programming and Performance Portability. 2017.

SYCL

OpenCL

```
const char *src =
     "__kernel void vecadd(global int *A,\n"
     " global int *B,\n" " global int *C) {\n"
     " size_t gid = get_global_id(0);\n"
     " C[gid] = A[gid] + B[gid];\n"
     "}"

clSetKernelArg(k, 0, sizeof(cl_mem), &ABuf);
clSetKernelArg(k, 1, sizeof(cl_mem), &BBuf);
clSetKernelArg(k, 2, sizeof(cl_mem), &CBuf);

clEnqueueNDRangeKernel(q, k, 1, NULL, {SIZE}, {32, 1, 1}, 0,
     NULL, NULL);
```

SYCL

```
auto A = ABuf.get_access(cgh);
auto B = BBuf.get_access(cgh);
auto C = CBuf.get_access(cgh);

cgh.parallel_for(
      cl::sycl::range(CBuf.size()),
      [=](cl::sycl::id idx) {
            C[idx] = A[idx] + B[idx];
});
```

Source: Alastair Murray. Codeplay. Layering Abstractions, Heterogeneous Programming and Performance Portability. 2017.

References

- ToUCH: Teaching Undergrads Collaborative and Heterogeneous Computing in Consortium for Computing Sciences in Colleges South Central Conference (CCSC19), 2019
- Alastair Murray. Codeplay. Layering Abstractions, Heterogeneous Programming and Performance Portability. 2017.