

OpenCL and Multi-GPU



OpenCL

CPUs
Multiple cores driving
performance increases

GPUs
Increasingly general
purpose data-parallel
computing

Emerging
Intersection

OpenCL
Heterogeneous
Computing

Graphics
APIs and
Shading
Languages

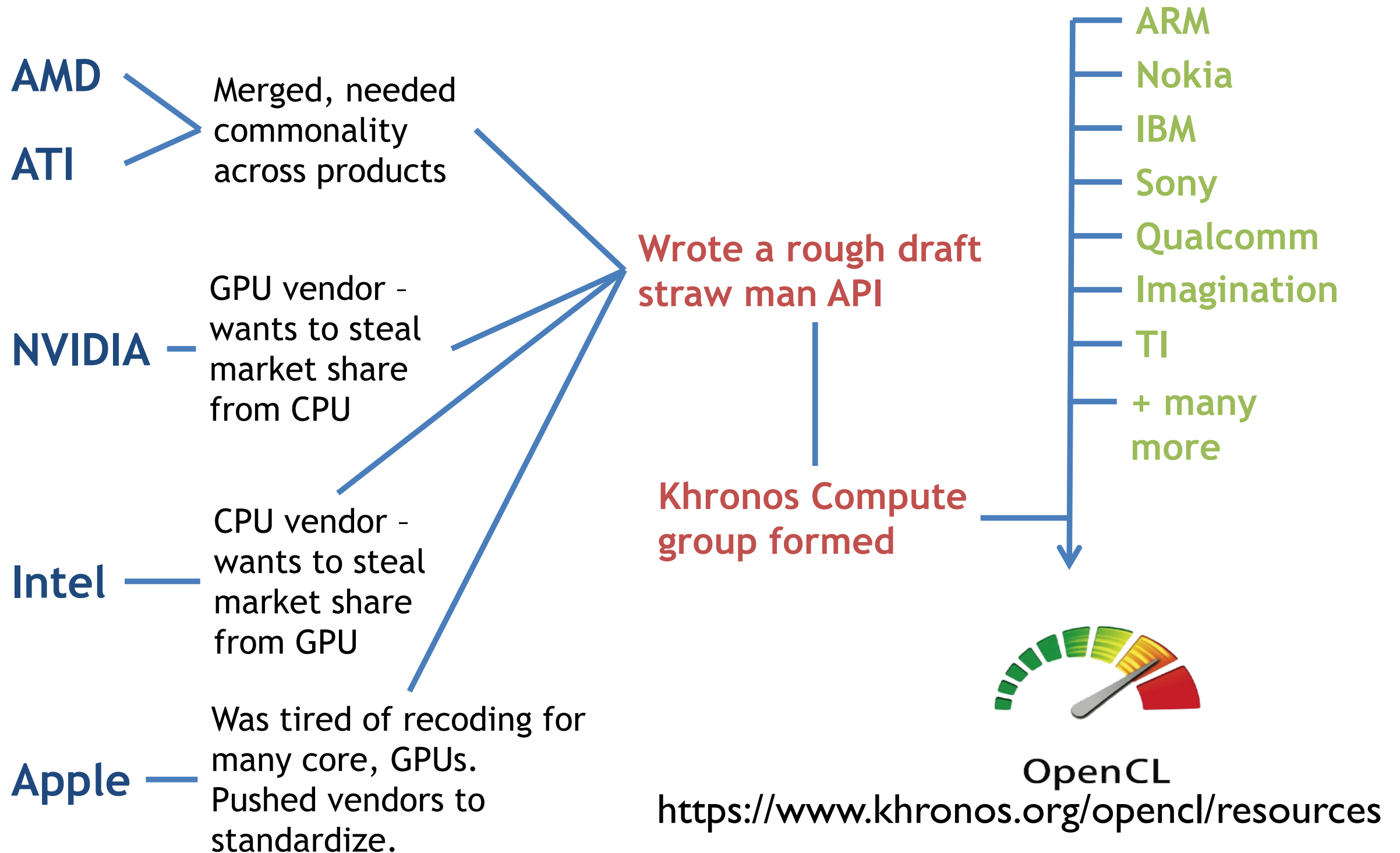
Multi-
processor
programming -
e.g. OpenMP

cheatsheet



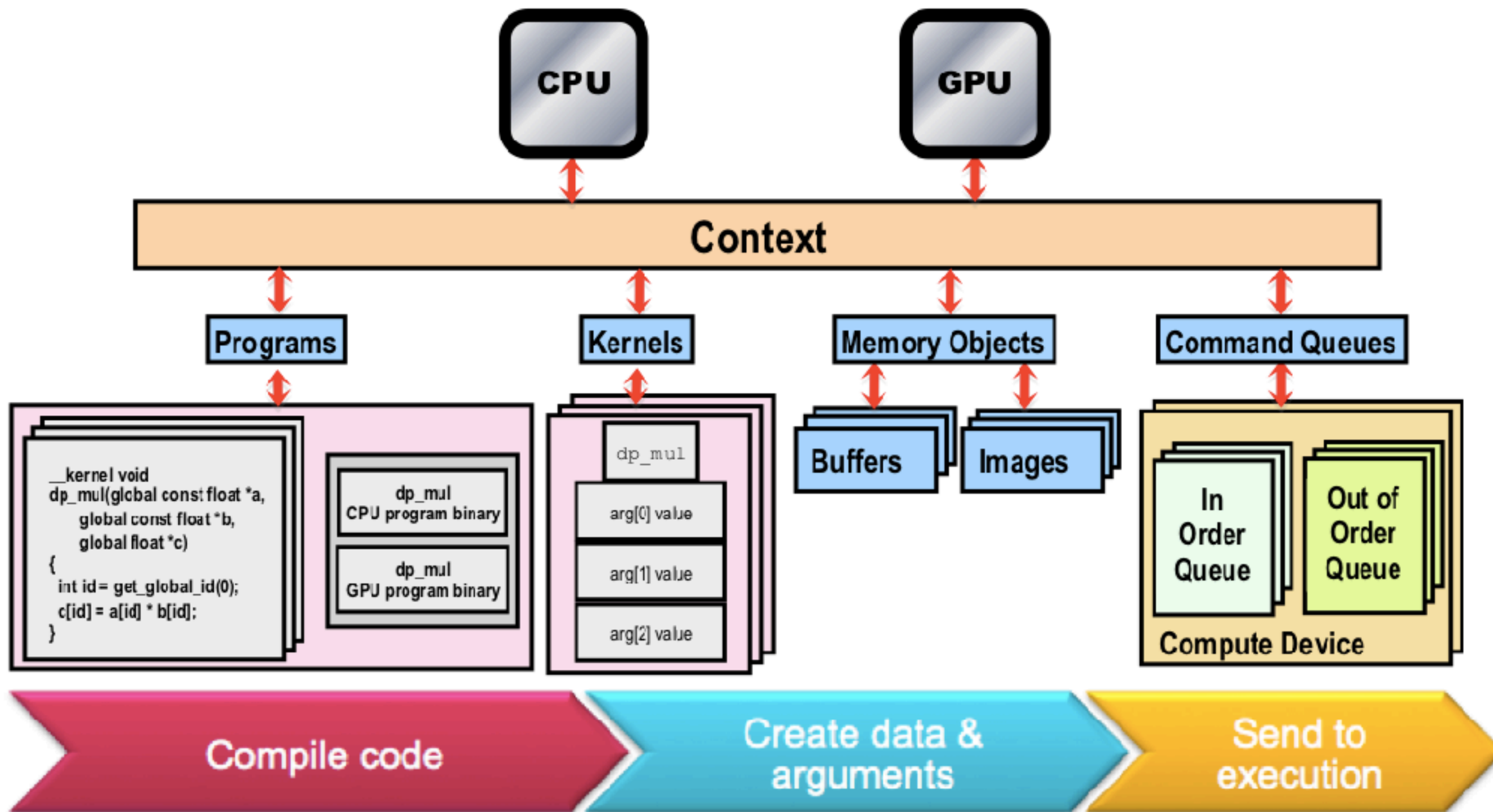
<https://www.khronos.org/files/ocl-1-1-quick-reference-card.pdf>

The origins of OpenCL



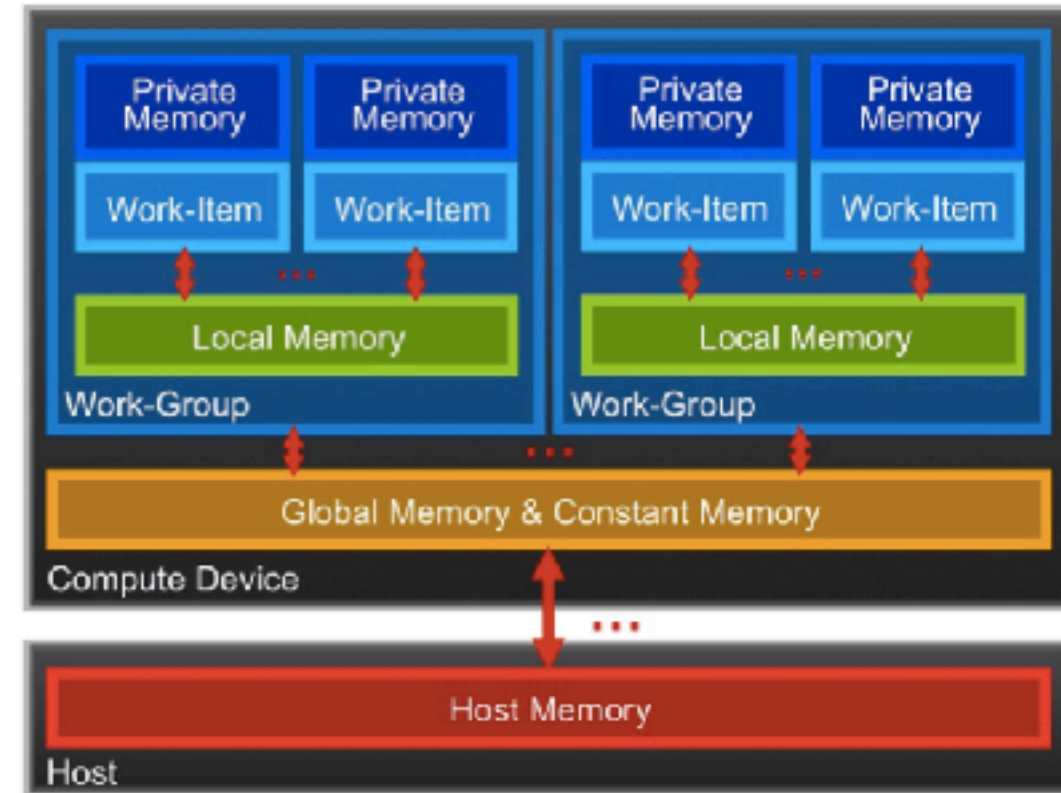
Third party names are the property of their owners.

OpenCL execution model



OpenCL

- **Private Memory**
 - Per work-item
- **Local Memory**
 - Shared within a work-group
- **Global/Constant Memory**
 - Visible to all work-groups
- **Host memory**
 - On the CPU



Bandwidths

Private memory
 $O(2-3)$ words/cycle/WI

Local memory
 $O(10)$ words/cycle/WG

Global memory
 $O(100-200)$ GBytes/s

Host memory
 $O(1-100)$ GBytes/s

Sizes

Private memory
 $O(10)$ words/WI

Local memory
 $O(1-10)$ KBytes/WG

Global memory
 $O(1-10)$ GBytes

Host memory
 $O(1-100)$ GBytes

00_platform.cpp

```
#include <cstdio>
#include <CL/cl.h>

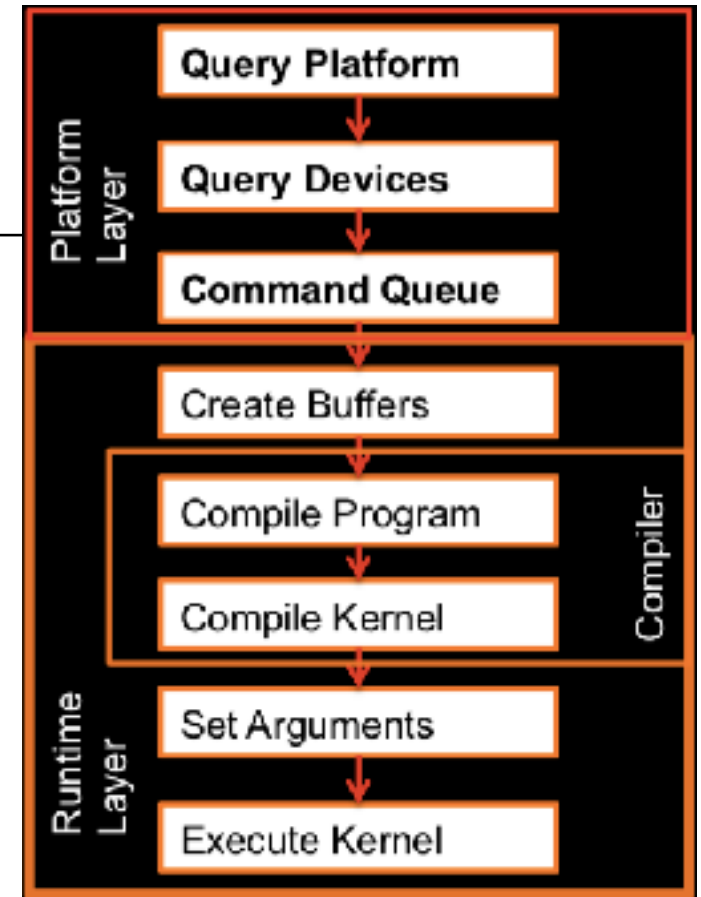
int main() {
    cl_platform_id *platformId = new cl_platform_id [3];
    cl_uint platformCount;
    clGetPlatformIDs(3, platformId, &platformCount);
    char *name = new char [32];
    char *version = new char [32];
    for (int i=0; i<platformCount; i++) {
        clGetPlatformInfo(platformId[i], CL_PLATFORM_NAME, 32, name, NULL);
        clGetPlatformInfo(platformId[i], CL_PLATFORM_VERSION, 32, version, NULL);
        printf("Platform %d: %s, %s\n", i, name, version);
    }
    delete[] name;
    delete[] version;
    delete[] platformId;
}
```

01_hello.cpp

```
#include <stdio>
#include <CL/cl.h>

const char *source = "__kernel void hello() {\n\
    printf(\"Hello GPU\\n\\n\"); }";

int main() {
    cl_platform_id platform;
    cl_device_id device;
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    cl_command_queue commandQueue = clCreateCommandQueue(context, device, 0, NULL);
    cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
    cl_build_program(program, 1, &device, NULL, NULL, NULL);
    cl_kernel kernel = clCreateKernel(program, "hello", NULL);
    printf("Hello CPU\\n");
    size_t global = 1, local = 1;
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
    clFinish(commandQueue);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(commandQueue);
    clReleaseContext(context);
}
```



02_memcpy.cpp

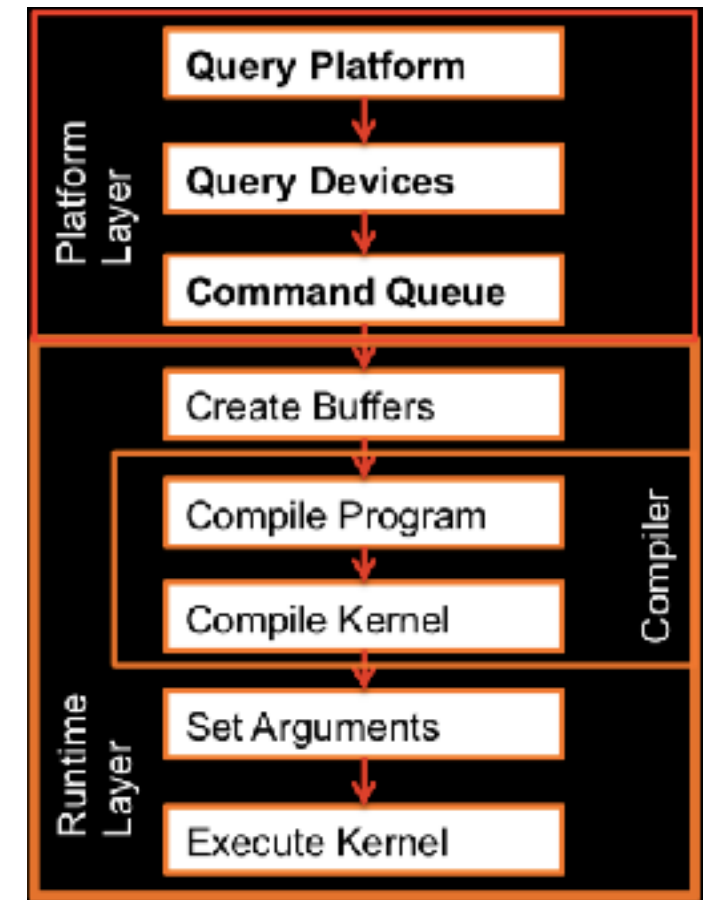
```
#include <stdio>
#include <CL/cl.h>

const char *source = "__kernel void memcpy(__global float* a) {\n\
    \"int i = get_global_id(0);\"\n\
    \"a[i] = i;}\";

int main() {
    cl_platform_id platform;
    cl_device_id device;
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    cl_command_queue commandQueue = clCreateCommandQueue(context, device, 0, NULL);
    cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
    cl_build_program(program, 1, &device, NULL, NULL, NULL);
    cl_kernel kernel = clCreateKernel(program, "memcpy", NULL);

    int size = 4 * sizeof(float);
    cl_mem a = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, NULL, NULL);
    float *b = (float*) malloc(size);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &a);
    size_t global = 4, local = 2;
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
    clEnqueueReadBuffer(commandQueue, a, CL_TRUE, 0, size, b, 0, NULL, NULL);
    for (int i=0; i<4; i++) printf("%f\\n",b[i]);
    clReleaseMemObject(a);
    delete[] b;

    clFinish(commandQueue);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(commandQueue);
    clReleaseContext(context);
}
```

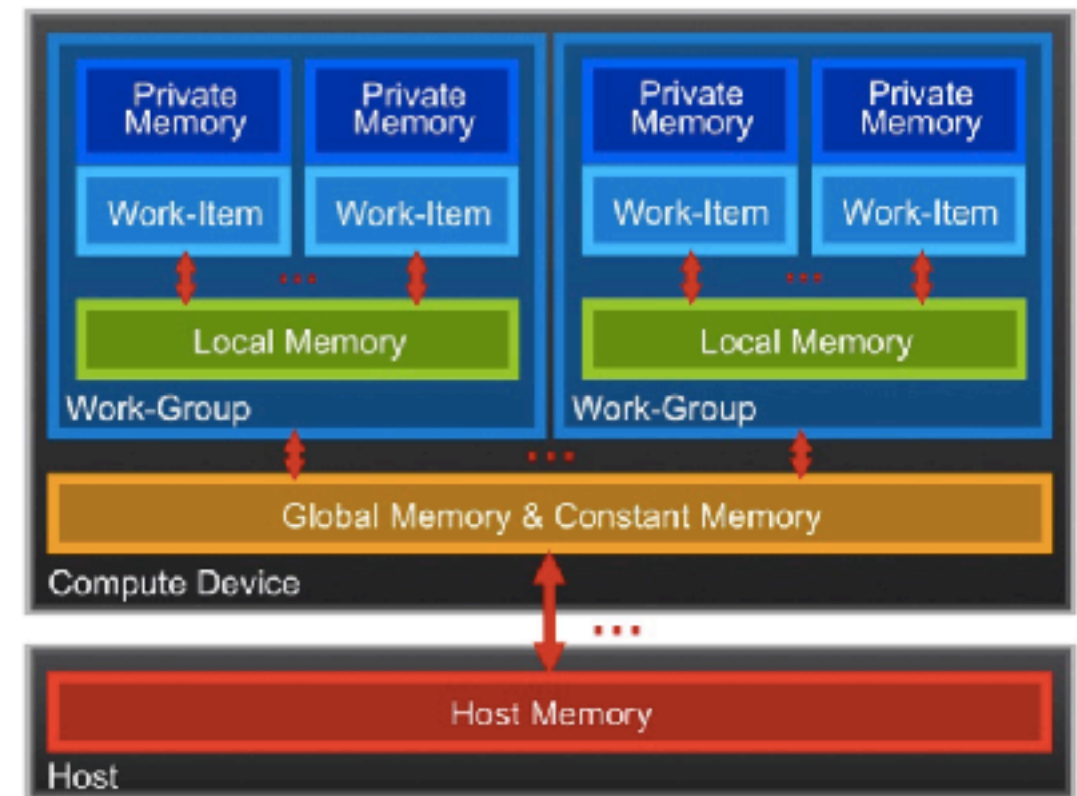


03_local.cpp

```
#include <stdio>
#include <CL/cl.h>

const char *source = "__kernel void memcpy(__global float* a) {\n\
    \"__local float b[2];\"\n\
    \"barrier(CLK_LOCAL_MEM_FENCE);\"\n\
    \"b[get_local_id(0)] = 10 * get_group_id(0) + get_local_id(0);\"\n\
    \"barrier(CLK_LOCAL_MEM_FENCE);\"\n\
    \"a[get_global_id(0)] = b[get_local_id(0)];}\"";
```

- **Private Memory**
 - Per work-item
- **Local Memory**
 - Shared within a work-group
- **Global/Constant Memory**
 - Visible to all work-groups
- **Host memory**
 - On the CPU



04_sum.cpp

```
const char *source = "__kernel void memcpy(__global int* a, __global int* sum) {\n    \"atomic_add(sum,a[get_global_id(0)]);}\"";

int main() {
    cl_platform_id platform;
    cl_device_id device;
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    cl_command_queue commandQueue = clCreateCommandQueue(context, device, 0, NULL);
    cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
    clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    cl_kernel kernel = clCreateKernel(program, "memcpy", NULL);

    int size = 4 * sizeof(int);
    cl_mem a = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, NULL);
    int *b = (int*) malloc(size);
    for (int i=0; i<4; i++) b[i] = 1;
    cl_mem sum = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(int), NULL, NULL);
    int *c = (int*) malloc(sizeof(int));
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &a);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &sum);
    size_t global = 4, local = 2;
    clEnqueueWriteBuffer(commandQueue, a, CL_TRUE, 0, size, b, 0, NULL, NULL);
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
    clEnqueueReadBuffer(commandQueue, sum, CL_TRUE, 0, sizeof(int), c, 0, NULL, NULL );
    printf("%d\\n\",*c);
    clReleaseMemObject(a);
    clReleaseMemObject(sum);
    delete[] b;
    delete[] c;

    clFinish(commandQueue);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(commandQueue);
    clReleaseContext(context);
}
```

mpi/00_init.cpp

```
#include "mpi.h"
#include <cstdio>
int main(int argc, char ** argv) {
    MPI_Init(&argc, &argv);
    int mpisize, mpirank;
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    printf("rank: %d/%d\n", mpirank, mpisize);
    MPI_Finalize();
}
```

```
> mpicxx 00_init.cpp
> mpirun -np 2 ./a.out
```

cuda/00_hello.cu

```
#include <cstdio>

__global__ void mykernel(void) {
}

int main() {
    mykernel<<<1,1>>>();
    printf("Hello CPU\n");
    return 0;
}
```

> nvcc 00_hello.cu

> ./a.out

cuda-mpi/00_hello.cu

```
#include <mpi.h>
#include <cstdio>

__global__ void mykernel(void) {
}

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int mpisize, mpirank;
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    mykernel<<<1,1>>>();
    printf("rank: %d/%d\n",mpirank,mpisize);
    MPI_Finalize();
}
```

CUDA+MPI

```
> mpicxx 00_hello.cu
```

```
00_hello.cu: file not recognized: File format not recognized
```

```
> nvcc 00_hello.cu
```

```
00_hello.cu:1:17: error: mpi.h: No such file or directory
```

```
> export CPATH=$CPATH:/usr/apps.sp3/isv/intel/ParallelStudioXE/  
ClusterEdition/2016-Update3/compilers_and_libraries_2016.3.210/  
linux/mpi/intel64/include
```

```
> nvcc 00_hello.cu
```

```
/tmp/tmpxft_000066d1_00000000-17_step02.o: In function `main':
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x165): undefined reference to `MPI_Init'
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x173): undefined reference to `MPI_Comm_size'
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x181): undefined reference to `MPI_Comm_rank'
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x191): undefined reference to
```

```
`MPI_Get_processor_name'
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x1d8): undefined reference to `MPI_Barrier'
```

```
tmpxft_000066d1_00000000-4_step02.cudafe1.cpp:(.text+0x28b): undefined reference to `MPI_Finalize'
```

```
> export LIBRARY_PATH=$LIBRARY_PATH:/usr/apps.sp3/isv/intel/  
ParallelStudioXE/ClusterEdition/2016-Update3/  
compilers_and_libraries_2016.3.210/linux/mpi/intel64/lib
```

```
> nvcc 00_hello.cu -lmpi
```

01_device.cu

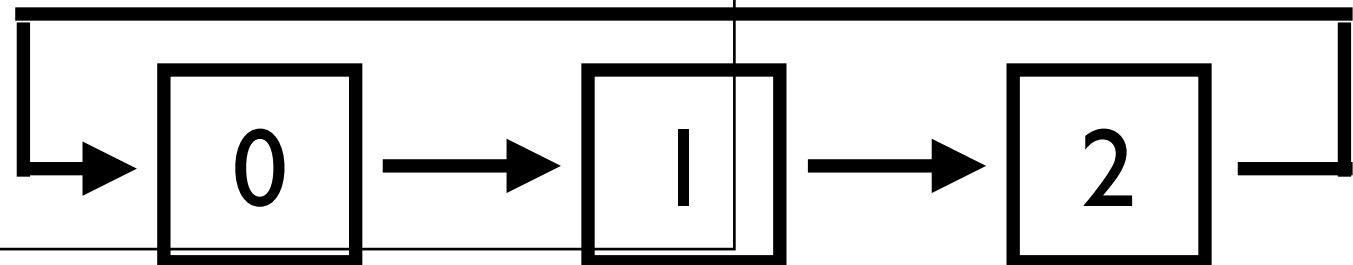
```
#include <mpi.h>
#include <stdio.h>

__global__ void GPU_Kernel() {
    printf(" GPU block   : %d / %d  GPU thread : %d / %d\n",
           blockIdx.x, gridDim.x, threadIdx.x, blockDim.x);
}

int main(int argc, char **argv) {
    int mpisize, mpirank, gpusize, gpurank;
    cudaGetDeviceCount(&gpusize);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    cudaSetDevice(mpirank % gpusize);
    cudaGetDevice(&gpurank);
    for (int irank=0; irank!=mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("MPI rank      : %d / %d  GPU device : %d / %d\n",
                   mpirank, mpisize, gpurank, gpusize);
            GPU_Kernel<<<2,2>>>>();
            cudaThreadSynchronize();
        }
    }
    MPI_Finalize();
}
```


02_send_recv.cu

```
#include <mpi.h>
#include <stdio>
__global__ void GPU_Kernel(int *send) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    send[i] += 10 * i;
}
int main(int argc, char **argv) {
    int mpisize, mpirank;
    int size = 4 * sizeof(int);
    int *send = (int *)malloc(size);
    int *recv = (int *)malloc(size);
    int *d_send, *d_recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    for(int i=0; i<4; i++)
        send[i] = mpirank;
    cudaSetDevice(mpirank % mpisize);
    cudaMalloc((void **) &d_send, size);
    cudaMalloc((void **) &d_recv, size);
    cudaMemcpy(d_send, send, size, cudaMemcpyHostToDevice);
    GPU_Kernel<<<2,2>>>>(d_send);
    cudaMemcpy(send, d_send, size, cudaMemcpyDeviceToHost);
    int sendrank = (mpirank + 1) % mpisize;
    int recvrank = (mpirank - 1 + mpisize) % mpisize;
    MPI_Send(send, 4, MPI_INT, sendrank, 0, MPI_COMM_WORLD);
    MPI_Recv(recv, 4, MPI_INT, recvrank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int irank=0; irank<mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("rank%d: send_rank=%d, recv_rank=%d\n", mpirank, sendrank, recvrank);
            printf("send=[%d %d %d %d], recv=[%d %d %d %d]\n",
                send[0],send[1],send[2],send[3],recv[0],recv[1],recv[2],recv[3]);
        }
    }
    free(send); free(recv);
    cudaFree(d_send); cudaFree(d_recv);
    MPI_Finalize();
}
```

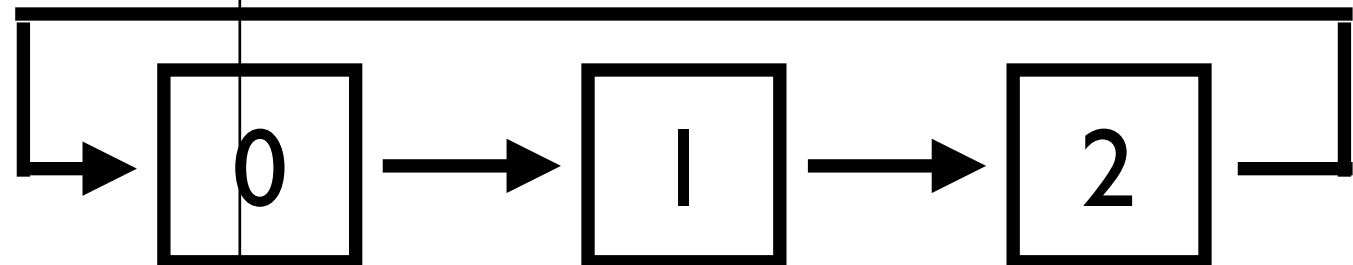


03_isend_irecv.cu

```
#include <mpi.h>
#include <stdio>
__global__ void GPU_Kernel(int *send) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    send[i] = 1;
}

#define N (2048*2048)
#define M 512

int main(int argc, char **argv) {
    int mpisize, mpirank;
    int size = N * sizeof(int);
    int *send = (int *)malloc(size);
    int *recv = (int *)malloc(size);
    int *d_send, *d_recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    cudaSetDevice(mpirank % mpisize);
    cudaMalloc((void **) &d_send, size);
    cudaMalloc((void **) &d_recv, size);
    cudaMemcpy(d_send, send, size, cudaMemcpyHostToDevice);
    GPU_Kernel<<<N/M,M>>>>(d_send);
    cudaMemcpy(send, d_send, size, cudaMemcpyDeviceToHost);
    int sendrank = (mpirank + 1) % mpisize;
    int recvrank = (mpirank - 1 + mpisize) % mpisize;
    MPI_Request reqs[2];
    MPI_Status stats[2];
    MPI_Isend(send, N, MPI_INT, sendrank, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(recv, N, MPI_INT, recvrank, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Waitall(2, reqs, stats);
    int sum = 0;
    for (int i=0; i<N; i++)
        sum += recv[i];
    for (int irank=0; irank<mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("rank%d: sum=%d, N=%d\n", mpirank, sum, N);
        }
    }
    free(send); free(recv);
    cudaFree(d_send); cudaFree(d_recv);
    MPI_Finalize();
}
```

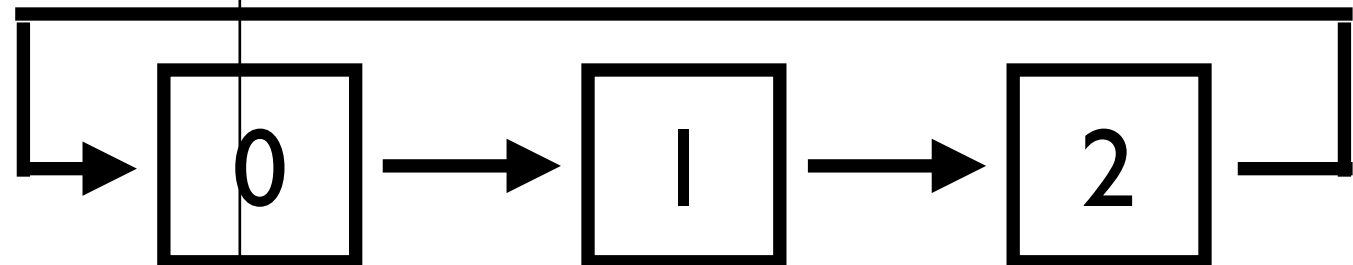


03_isend_irecv.cu

```
#include <mpi.h>
#include <stdio>
__global__ void GPU_Kernel(int *send) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    send[i] = 1;
}

#define N (2048*2048)
#define M 512

int main(int argc, char **argv) {
    int mpisize, mpirank;
    int size = N * sizeof(int);
    int *send = (int *)malloc(size);
    int *recv = (int *)malloc(size);
    int *d_send, *d_recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    cudaSetDevice(mpirank % mpisize);
    cudaMalloc((void **) &d_send, size);
    cudaMalloc((void **) &d_recv, size);
    cudaMemcpy(d_send, send, size, cudaMemcpyHostToDevice);
    GPU_Kernel<<<N/M,M>>>>(d_send);
    cudaMemcpy(send, d_send, size, cudaMemcpyDeviceToHost);
    int sendrank = (mpirank + 1) % mpisize;
    int recvrank = (mpirank - 1 + mpisize) % mpisize;
    MPI_Request reqs[2];
    MPI_Status stats[2];
    MPI_Isend(send, N, MPI_INT, sendrank, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(recv, N, MPI_INT, recvrank, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Waitall(2, reqs, stats);
    int sum = 0;
    for (int i=0; i<N; i++)
        sum += recv[i];
    for (int irank=0; irank<mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("rank%d: sum=%d, N=%d\n", mpirank, sum, N);
        }
    }
    free(send); free(recv);
    cudaFree(d_send); cudaFree(d_recv);
    MPI_Finalize();
}
```

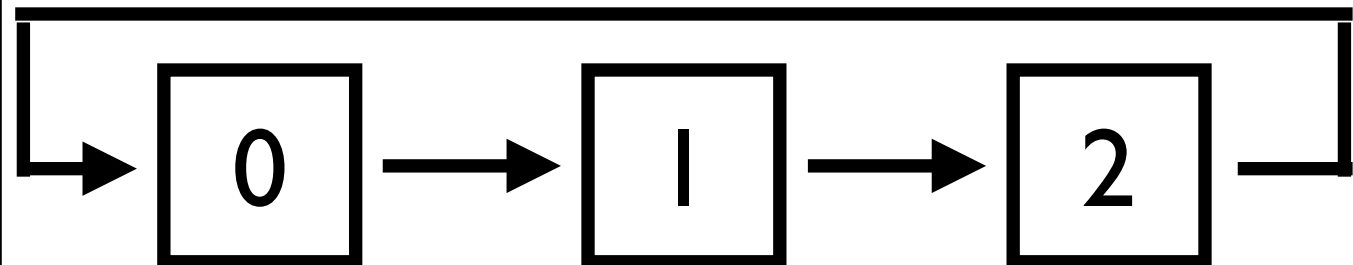


04_isend_irecv.cu

```
#include <mpi.h>
#include <stdio>
#include <sys/time.h>
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return double(tv.tv_sec)+double(tv.tv_usec)*1e-6;
}
__global__ void GPU_Kernel(int *send) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    send[i] = 1;
}

#define N (2048*2048)
#define M 512

int main(int argc, char **argv) {
    int mpisize, mpirank;
    int size = N * sizeof(int);
    int *send = (int *)malloc(size);
    int *recv = (int *)malloc(size);
    int *d_send, *d_recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    cudaSetDevice(mpirank % mpisize);
    cudaMalloc((void **) &d_send, size);
    cudaMalloc((void **) &d_recv, size);
    double tic = get_time();
    cudaMemcpy(d_send, send, size, cudaMemcpyHostToDevice);
    double toc = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "Memcpy host to device", toc-tic);
    GPU_Kernel<<<N/M,M>>>>(d_send);
    cudaThreadSynchronize();
    tic = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "CUDA kernel", tic-toc);
    cudaMemcpy(send, d_send, size, cudaMemcpyDeviceToHost);
    toc = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "Memcpy device to host", toc-tic);
    int sendrank = (mpirank + 1) % mpisize;
    int recvrank = (mpirank - 1 + mpisize) % mpisize;
    MPI_Request reqs[2];
    MPI_Status stats[2];
    MPI_Isend(send, N, MPI_INT, sendrank, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(recv, N, MPI_INT, recvrank, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Waitall(2, reqs, stats);
    tic = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "MPI communication", tic-toc);
    int sum = 0;
    for (int i=0; i<N; i++)
        sum += recv[i];
    for (int irank=0; irank<mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("rank%d: sum=%d, N=%d\n", mpirank, sum, N);
        }
    }
    free(send); free(recv);
    cudaFree(d_send); cudaFree(d_recv);
    MPI_Finalize();
}
```



05_unified.cu

```
#include <mpi.h>
#include <cmath>
#include <stdio.h>
#include <sys/time.h>
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return double(tv.tv_sec)+double(tv.tv_usec)*1e-6;
}
__global__ void GPU_Kernel(int *send) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    send[i] = 1;
}

#define N (2048*2048*2)
#define M 512

int main(int argc, char **argv) {
    int mpisize, mpirank;
    int size = N * sizeof(int);
    int *send, *recv;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
    cudaSetDevice(mpirank % mpisize);
    cudaMallocManaged(&send, size);
    cudaMallocManaged(&recv, size);
    double tic = get_time();
    //cudaMemcpy(d_send, send, size, cudaMemcpyHostToDevice);
    double toc = get_time();
    //if(mpirank==0) printf("%-25s : %lf s\n", "Memcpy host to device", toc-tic);
    GPU_Kernel<<<N/M,M>>>(send);
    cudaThreadSynchronize();
    tic = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "CUDA kernel", tic-toc);
    //cudaMemcpy(send, d_send, size, cudaMemcpyDeviceToHost);
    toc = get_time();
    //if(mpirank==0) printf("%-25s : %lf s\n", "Memcpy device to host", toc-tic);
    int sendrank = (mpirank + 1) % mpisize;
    int recvrank = (mpirank - 1 + mpisize) % mpisize;
    MPI_Request reqs[2];
    MPI_Status stats[2];
    MPI_Isend(send, N, MPI_INT, sendrank, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(recv, N, MPI_INT, recvrank, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Waitall(2, reqs, stats);
    tic = get_time();
    if(mpirank==0) printf("%-25s : %lf s\n", "MPI communication", tic-toc);
    for (int i=0; i<N; i++)
        sum += recv[i];
    for (int irank=0; irank<mpisize; irank++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (mpirank == irank) {
            printf("rank%d: sum=%d, N=%d\n", mpirank, sum, N);
        }
    }
    cudaFree(send); cudaFree(recv);
    MPI_Finalize();
}
```

