Parallel programming with CUDA and MPI

D. Bikov*, M. Pashinska ** and N. Stojkovikj*

* Faculty of computer science, "Goce Delcev" University, Stip, Republic of Macedonia

** Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Veliko Tarnovo, Bulgaria
dusan.bikov@ugd.edu.mk, mariqpashinska@math.bas.bg, natasa.stojkovik@ugd.edu.mk

Abstract – Nowadays parallel programming is stepping up on the big door and slowly surpasses the traditional sequential programming model. The main idea here is to show how it can combine two parallel programming models in order to use effectively all available computing resources. Parallel programming models of interest here are MPI and CUDA. By combining both models we could use potentially all available processing resources.

I. INTRODUCTION

Modern computer systems consist of more and more parallel processing resources which is due to the latest achievements in computing technologies. On the other hand, to be able to use these parallel processing capabilities we need to imply a suitable programming model. Nowadays computing systems combine two main computation resources with significantly pronounced parallel computing capabilities. The main building component of every computer system is Central Processing Unit (CPU) which is low latency oriented while the other significant component represents Graphic Processing Units (GPU) which is throughput oriented and owns massive parallel processing capability.

Trends in computing technologies will inevitably introduce high education Computer Science subjects that cover parallel computing and programming. The strength of the CPU is in the efficient low latency-oriented design. They contain few cores and can handle few threads at a time. On the other hand, the recent GPU have high throughput-oriented design and are composed from thousands of cores that can handle execution of thousands of threads simultaneously. Combining MPI (Message Passing Interface) and CUDA (Compute Unified Device Architecture) programming models allows utilization of all available computation resources. Through the short guide we will show how it is possible to achieve that. Building the application which will combine those two programming models could ensure utilization of the whole computation resources available in one computer system. Knowledge about this issue only can help the students easily to step forward and to acquire new skills and learn new and highly demanded modern technologies. Modern multicore computer systems brought parallel computing to wide use general-purpose PC, embedded system, game consoles, smart phones, smart TV etc.

Because of the educational purpose of this paper we will focus on giving step by step guidelines for combining MPI and CUDA programming models by exploiting initial programming examples. Recommended prerequisites that are needed before starting to write combined MPI and CUDA programs are a knowledge of C/C++, knowledge of computer architectures and operating systems. It is not necessary to have knowledge of computer graphics or parallel programming. There are many factors for writing efficient programs that will help exploiting all computation resources. Here we will cover some basics without going deeper or showing optimizations techniques for writing efficient parallel programming code.

Growing computing application demand naturally leads to change from traditional sequential computation to more promising parallel computing. To fill this gap there is a need for massive parallel processing capability units. Latest GPU have a highly parallel structure that makes them more effective for algorithms where processing of large blocks of data is done in parallel [1] [2]. With the passing of time GPU have surpassed the CPU in many ways. Improvements of the GPU go side by side with a growing range of applications from the traditional computation, signal processing, irregular computations to Machine Learning, Deep Learning, AI, Computer Vision, Supercomputing and more. It is very important for students to get familiar with GPU technologies. However, combining programming models that ensures usage of the whole available computation resources is of a completely different magnitude.

[1]The research of the first author was supported by Bulgarian Science Fund under Contract DN-02-2/13.12.2016.

[2]The research of the second author was supported, in part, by the Bulgarian Ministry of Education and Science by Grant No. DO1-221/03.12.2018 for NCHDC, a part of the Bulgarian National Roadmap on RIs.

Various approaches and techniques exist for combining CPU and GPU resources. Most of the approaches are intended for computer cluster systems while others are for writing programs intended for heterogeneous platforms. These approaches can be separated into specific programming languages, API (Application Programming Interface) and frameworks [3]. There are also techniques for reclaiming lost performance and inefficient resources use. Some of them examine GPU utilization of individual kernels and design algorithmic techniques for maximizing resource utilization. Our intentions are to show a general method for utilizing simultaneously CPU and GPU resources and to achieve this MPI and CUDA programming models are used.

The paper is organized as follows. The general principles of MPI and CUDA programming models are given in section 2. Section 3 is devoted to describing the possibility for combining both programming models in a single program. Initial programming examples and compile instructions are presented in section 4. A few conclusion sentences are given in the end.

II. CPU AND GPU COMPUTING MODEL

One of the main differences between CPU and GPU computing models is how they execute tasks. The CPU is optimized for sequential execution while the GPU can execute thousands of tasks simultaneously.

A. CPU computing with MPI

MPI is a portable message-passing standard API designed by a group of researchers from academia and industry and is intended for a wide variety of parallel architectures. It is a standard for data communication via messages between distributed processes and is often used in HPC (High Performance Computing) for building scale applications on computer clusters. There are several well tested and efficient implementations of MPI that are fully compatible with CUDA, CUDA Fortran, and OpenACC designed for parallel computing. There are several CUDA-aware MPI open source and commercial implementations and some of them are MVAPICH2, OpenMPI, CRAY MPI, IBM Platform MPI, SGI MPI. There are a bunch of reasons for writing MPI and CUDA combined parallel programming code. Depending on the hardware or the problem that needs to be solved, the reasons for using these parallel programming approaches may vary. This approach can be applied if there is a problem with very large data size that needs to fit in the memory of a single GPU. Another reason is enabling multi-GPU applications to scale across multiple nodes. Our reason is accelerating an existing sequential application in order to achieve more efficient use of all available computing resources.

MPI standard defines syntax and semantics of library routines used for writing a wide range of portable message passing programs in C, C++, and Fortran. Other languages can also interface with such libraries. Parallel programs that use MPI can consist of separate processes, each with its own address space in which it is run. Each MPI process has its own rank and for the duration of the program execution there are fixed number of ranks executing the same program. It facilitates the use of SPMD (Single Program, Multiple Data) [4] programming model but it is not required, there are MPI implementations that allow multiple, different, executables to be started in the same MPI job. MPI process rank runs on a different core and own private memory and executes instructions at its own rate. The ranks can copy or move data between private memories via a shared interconnection. The communication can be performed by point-to-point (send/receive) which is communication between two processes, or by collective communication among the group of processes. In general, all ranks perform the same activity - compute or communicate at the same time. It should be noted that ranks workloads are not well balanced. It is important to understand message passing. They are like email with a destination and message body, which can be empty. Communication is bidirectional and requires explicit sender and receiver participation. The messages provide two services as memory to memory copy across address spaces and 2-sided handshake synchronization. If multiple messages are sent to the same destination from the same rank, then the messages will be received in the same order. But if different ranks send messages to the same destination, the order of receipt is not defined across sources. For writing message passing programs a library called MPI is used. There are a few releases of this library and the first one MPI-1 is from 1994. The first release contains 125 routines and there are more than 430 routines in MPI-3. There are at least six routines needed for the most MPI programs: start, end, query MPI execution state, point-to-point message passing. The library has additional tools for launching the MPI program (*mpirun*) and daemon which moves the data across the network.

B. GPU computing with CUDA

CUDA is powerful parallel computing platform created by Nvidia and it allows software developers to use a CUDA-enabled GPU [5] for general purpose

computing. This platform allows developers to directly interact with the GPU resources and harness their power for writing efficient parallel programs. The GPU programs contain two parts, there is a control (sequential) part that is executed by a single CPU thread and there is a parallel GPU executed part that runs thousands threads in parallel on as many cores as possible at each moment. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. It also supports other computational interfaces such as OpenCL, OpenGL, C++ AMP, and the third-party wrappers are available for Python, Julia, MATLAB, etc. Through the years Nvidia developed different micro-architecture for the various GPU. Depending on the microarchitecture generally Nvidia GPU are organized in SM (Streaming Multiprocessor) with a set of registers cache for constants, texture cache, shared memory (L1 cache) and global memory. Each SM consists of a number of SP (Streaming Processor), and SFU (Special Function Unit) used for transcendental functions. Common name for SP is CUDA core. The SP contains several ALU (Arithmetic Logic Unit) and FPU (Floating Point Unit). Execution model used by the SM is SIMT (Single-Instruction Multiple-Threads) [6] which is similar to the SIMD (Single Instruction Multiple Data) by Flynn's taxonomy [7] of computer architectures classification. The communication between SM is performed through global memory.

CUDA C is essentially a C/C++ programming language with extensions that allow executing of parallel functions on GPU. The CUDA source code consists of a mixture of conventional C/C++ host code and GPU device functions. There is CUDA C compiler *nvcc* that separates the parallel (device) functions from the code. According to this on the top level of the CUDA application there is a master process that runs on the CPU. This process is responsible for data flow between main memory and GPU memory. This process performs several tasks such as GPU initialization, allocation of main and GPU memory, moving data between main to GPU memory, launching of kernels (functions) on the GPU, fetching back processed data, deallocation of the memory and termination.

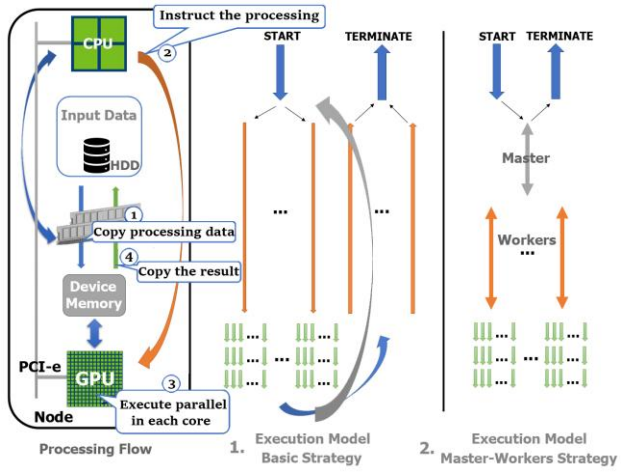

Figure 1. Example of processing flow and execution strategy model

III. BASIC PARALLEL PROGRAMMING STRATEGIES

The common problem in parallel programming is balancing of the computational load among a set of parallel processing resources. It is especially important to use the appropriate parallel programming strategy. The choice of suitable parallel programming strategy highly depends on the problem itself. In this section two widely accepted parallel programming strategies will be presented. These programming strategies are suitable for task parallel programs with no communication between tasks. It is possible to have communications between the tasks. However, it is recommended to be infrequent to reduce negative consequences on efficiency. The two typical strategies will be explained here. The structure of the programs is simple and has several MPI processes that operate with the same GPU. If there are multiple GPU, the MPI process can handle all of them. It is widely known that this programming structure introduces contest switch overheads. The MPI process is handling the main memory while CUDA kernels update the GPU memory.

In order to explain the building of a combined MPI and CUDA program step by step it is important to introduce the parallel programming strategies. On Fig. 1 is shown the general structure of processing flow together with the execution of the strategies.

A. Basic Parallel Strategy Model

The Basic Parallel Strategy Model is the first strategy shown on Fig.1 and is marked by 1. As it is implied by the name, this strategy has typical basic characteristics of a simple bound MPI and CUDA program. This strategy presents a simple solution for building MPI and CUDA programs. As it can be seen from Fig 1. all MPI processes run simultaneously and can start CUDA kernel function on the GPU. From Fig. 1 it can be noticed that there is circular execution on MPI – CUDA threads. It is important to have balanced work distribution through the MPI – CUDA threads. Balanced work distribution ensures efficient computation resource use and better computing performance. As already mentioned, there is a master process that runs on the CPU which is responsible for initialization, allocation and computation performing on GPU. In

TABLE I. DESCRIPTION OF THE TEST PLATFORM

Environment	Platform 1	Platform 2
CPU	Intel i7-8565U, 1.80GHz	Intel Xeon E5-2640, 2.50GHz
Memory	8 GB DDR4 2400 MHz	48 GB DDR3 1333 MHz
GPU	GeForce MX150	Nvidia TITAN X (Pascal)
OS	Ubuntu 16.04 LTS 64-bit	Ubuntu 18.04.3 LTS 64-bit
Compiler	gcc 5.4.0	gcc 7.4.0
CUDA compilation tools	10.1	9.1
GPU Driver	V416.56	V430.50
MPI	(Open MPI) 1.10.2	(Open MPI) 3.3a2

this case there are few MPI process instances that run on the CPU and they are responsible to ensure performance of all essential tasks simultaneously and independently. Because of the use of the same hardware resources there is an introduction of latency, resource contest, waiting, overheads and slower bandwidth. With the increase of the number of processes the negative effects get more pronounced. It is important to have balance between the number of processes and the scale of the problem. The efficiency of the parallel computation is directly connected to available hardware resources.

B. Master-Workers Parallel Strategy Model

Second strategy that will be described is the Master-Workers Parallel Strategy Model shown on Fig. 1 marked by 2. The effective solution by automatic dynamic load balancing is to define the single master process to manage collection of the tasks and collect the results. Then the set of process workers grab a task, compute the task, send the results back to the master and then grab the next task. This action proceeds until completion of all the tasks. The master process in this case is one MPI process that schedules computational tasks to other MPI worker processes. The worker process is behaving as a top-level process that maintains a CUDA instance. Any worker task is recommended to be with equal computing demand. This way it ensures efficiency and better computing performances. In this case the worker MPI processes run on the CPU and they are responsible for performing all essential tasks simultaneously and independently. The same hardware resources are used as in the first strategy, but there is also an extra master process. Because of this there are negative consequences as input query, task waiting time, and not equal distribution of tasks between the workers. The negative consequences are more expressed with the increase of the number of the processes. There needs to be a balance between hardware resources, process number and the scale of the computing problem.

IV. PARALLEL PROGRAM EXAMPLES

This section presents parallel program examples. The platforms that are used for testing the examples are described in Table 1. Platform 1, a graphic card NVIDIA GeForce 150MX [8], has 384 cores running at 1.53 GHz and 48 (GB/sec) memory bandwidth. Platform 2, a graphic card NVIDIA TITAN X [9], has 3584 cores running at 1.41 GHz and 480 (GB/sec) memory bandwidth.

CUDA-aware OpenMPI implementation is used for building a single MPI + CUDA program on Ubuntu OS. Open MPI can handle multiple GPU cards but in our case, there is only one GPU. The GPU card is utilized and shared between several MPI processes. Tests are performed on two diff erent classes of parallel processing capability hardware (Table I). It is important computing hardware and software to be compatible, update and properly configured.

A. Examples

The examples represent simple MPI and CUDA programs that are built according to the parallel strategies presented in the previous section. Here the examples represent the basic program skeleton structure of the explained strategies. One way to build a single MPI+CUDA program is to put both code MPI and CUDA code in a single file. This program can be compiled using nvcc, which internally uses gcc/g++ to compile your C/C++ code, and linked to your MPI library. Another way is to have MPI and CUDA code separate in two files, *main.c* and *example.cu* respectively.

The code from the first example is according to the Basic Parallel Strategy Model and contains the mentioned two files *main.c* and *examples.c*u. The *main.c*, containing the call to CUDA file, would look like:

```
#include <mpi.h>
#include <stdio.h>
//Function declaration
void call_kernel(...);

int main(int argc, char *argv[]) {
//variable declarations
int myrank, nProcs;
//Allocate memory
...
/*   Initialize   the   MPI   execution
environment. */
```

```
MPI_Init(argc, argv);
/* Get the number of MPI processes and
the rank of this process. */
MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
MPI_Comm_size(MPI_COMM_WORLD,&nProcs);
/* Call function 'call_kernel()' from
CUDA file example.cu */
call_kernel(...);
// Terminates MPI execution environment
MPI_Finalize();
//Free memory
...
}
```

In *example.cu*, the *call_kernel()* function is defined with the 'extern' keyword to make it accessible from *main.c*. The code it would look like:

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>
//CUDA kernel
__global__ void __kernel__(...)
{
 ... // Do some work
}
extern "C" void call_kernel(...)
{
//Load CPU data into GPU buffers
__kernel__<<<BlocksPerGrid,
ThreadsPerBlock >>>(...);
//Transfer data from GPU to CPU
//Free device memory
   }
```

The second example is based on the Master-Workers Parallel Strategy Model. It also consists of two files, *main.c* and *example.cu*. The *main.c*, containing the call to CUDA file, would look like:

```
#include <mpi.h>
#include <stdio.h>
#define ROOT 0
//message to processors
#define END_MSG 1
#define GET_TASK 2
#define NEW_TASK 3
#define NO_TASK 4
#define DONE 5

//Function declaration
void call_kernel(...);

int main(int argc, char **argv) {
// Variable declarations
int myRank, //id of current processor
    nprocs; //number of processors
int rc; // result from MPI operation
int dummy = 1; // fictive variable
...
//Allocate memory
...
/*  Initialize  the  MPI  execution
environment*/
MPI_Init(&argc, &argv);

/* Get the number of MPI processes and
the id rank of this process. */
MPI_Comm_size(MPI_COMM_WORLD,&nProcs);
MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
// Status of a reception operation
MPI_Status status;
if(myRank == 0) // Root
{
int hasWork = 15; // number of works

while(hasWork >= 0){
/*  Wait  message  (dummy)  from  free
processor (MPI_ANY_SOURCE)*/
rc= MPI_Recv(&dummy, 1, MPI_INT,
MPI_ANY_SOURCE, GET_TASK,
MPI_COMM_WORLD, &status);
// Master send work
rc = MPI_Send(&hasWork, 1, MPI_INT,
status.MPI_SOURCE, NEW_TASK,
MPI_COMM_WORLD);
hasWork--; //Decrement work variable
}
/* When while quit this means that no
more works and Root send message DONE
to all free processors */
int k;
for(k = 0; k < nprocs-1; k++) {
dummy = -100;
rc = MPI_Recv(&dummy, 1, MPI_INT,
MPI_ANY_SOURCE, GET_TASK,
MPI_COMM_WORLD, &status);
rc = MPI_Send(&dummy, 1, MPI_INT,
status.MPI_SOURCE, DONE,
MPI_COMM_WORLD);
} }
if(myRank != 0)// all other processors
{
//Must have end case, else not to work
while(1) {
int work, other;
dummy = myRank;
/* send to Root (0), that I am free,
and want some work */
rc = MPI_Send(&dummy, 1, MPI_INT, 0,
GET_TASK, MPI_COMM_WORLD);
//receive work from Root (int work)
rc = MPI_Recv(&work, 1, MPI_INT, 0,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// quit while
if(status.MPI_TAG == DONE){
break;
}
// Do some work from process
if(status.MPI_TAG == NEW_TASK) {
/* Call function 'call_kernel()' from
CUDA file example.cu */
```

```
call_kernel(...);
} } }
// Terminates MPI execution environment
MPI_Finalize();
}
```

The *example.cu*, contains the same code as Basic Parallel Strategy Model where *call_kernel()* is defined with the '*extern*' keyword to make it accessible from *main.c*.

B. Compile procedure

The described below way of building MPI+CUDA programs contains two files. These two files can be compiled using *mpicc*, and *nvcc* respectively into object files (.o) and combined into a single executable file using *mpicc*. Using *mpicc* means that the CUDA library must be linked.

The program can be build/executed in Debug and Release mode. Making object files, linking, building, and executing the programs were performed by using Ubuntu terminal and suitable commands. The compile instructions for build and execute in Debug mode, would look like:

```
mpicc -c main.c -o main.o
nvcc -c example.cu -o example.o
mpicc main.o example.o -lcudart -L
/usr/local/cuda-10.0/lib64/ -lstdc++ -o
mpicuda
```

It is necessary to carefully link the CUDA library. In the example above is shown linking the CUDA library executed on Platform 1 (see Table I). Compile instruction that look like:

```
mpiexec -n <numprocs> <program>
```

is one way to start `<program>` with the name of the program with an initial MPI_COMM_WORLD whose group contains `<numprocs>` number of processes. Example for request two processes to test the program would look like:

```
mpiexec -np 2 ./mpicuda
```

The compile instructions with additional optimization and customize options [10] [11] [12] for build and execute in Release mode, would look like:

```
mpicc -O3 -Wall -c -fmessage-length=0 -
MMD -MP -MF"main.d" -MT"main.d" -o
"main.o" "main.c"
nvcc -O3 --compile --relocatable-
device-code=false -gencode
arch=compute_50,code=compute_50 -
gencode arch=compute_50,code=sm_50 -
D_FORCE_INLINES -x cu -o  "example.o"
"example.cu"
mpicc main.o example.o -lcudart -L
/usr/local/cuda-10.1/lib64/ -lstdc++ -o
mpicuda
```

The compile instruction for executing the by request two processes to test the program would look like:

```
mpiexec -np 2 ./mpicuda
```

If there is a cluster with multiple CPU and GPU it can call different execution configuration. For example, it can request two processes and two GPUs to test the program by using PBS (Portable Batch System) script. Another way to execute the program on more than one GPU is expressly use of CUDA call *cudaSetDevice(number)* to set the current GPU, where *number* is GPU card ID (identifier).

V. CONCLUSIONS

In this paper, a short guideline is given for combining the two parallel programming approaches of MPI and CUDA. There is brief clarification of both parallel programming models. Through the two parallel programming strategies our intention is to get parallel programming closer to the students. In the last section the program skeleton of an example is shown. This example can be a starting point for building complex program structure. In the end of the section is shown how to compile and run the combined MPI and CUDA program.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

REFERENCES

[1] Kirk, D. B., Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier, 2013.

[2] Kurzak, J., D. A. Bader, J. Dongarra. Scientific Computing with Multicore and Accelerators. CRC Press, 2010.

[3] Mittal, S. and Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys (CSUR), 47(4), pp.1–35, 2015.

[4] Quinn Michael J.: Parallel Programming in C with MPI and OpenMP. 1st ed. McGraw-Hill Inc. 2004.

[5] Nvidia CUDA Home Page, https://developer.nvidia.com/cuda-zone. Last accessed 16 August 2020

[6] Lindholm E., J. Nickolls, S. Oberman, J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, Vol. 28, Issue 2, 2008.

[7] Flynn M., Flynn's Taxonomy. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA., pp. 689-697, 2011.

[8] NVIDIA GeForce 150MX Specification, https://www.geforce.com/hardware/notebook-gpus/geforce-mx150. Last accessed 14 April 2020

[9] NVIDIA GeForce TITAN X Specification, https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/. Last accessed 16 August 2020

[10] NVCC :: CUDA Toolkit Documentation, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Last accessed 12 September 2020

[11] GCC Command Options, https://gcc.gnu.org/onlinedocs/gcc-3.1.1/gcc/Invoking-GCC.html. Last accessed 12 September 2020

[12] FAQ: Compiling MPI applications, https://www.open-mpi.org/faq/?category=mpi-apps. Last accessed 12 September 2020