Al-Qadisiyah Journal of Pure Science
ISSN(Printed): 1997-2490 ISSN(Online): 2411-3514
DOI: 10.29350/jops
Al-Qadisiyah Journal of Pure Science

Programmatic Analysis Of Technologies And Systems (Graphics Processing Unit)

Yasir Khudheyer Abass Aloubade

Iraqi Ministry of Education, General Directorate of Education Baghdad Karkh3, Planning Department, Information and Communication Division .

ORCID: 0000-0001-9976-5406
yasirkhuiraq@gmail.com

Abstract

In the era of digital technology in our modern world, the rapid processing of large amounts of data is very important. Due to the frequent development and the amount of big data, there is an urgent need for high-performance computing.
One option to solve this problem is to introduce parallelism to computing systems, and use the power of graphic processing units (GPUs).
A GPU has thousands of cores that can perform similar operations independently. GPU calculations have become widespread in techniques such as neural networks, image processing, as well as in mathematical and financial calculations, where it is necessary to work with a large volume of computational data.
The purpose of the work is to compare two software technologies on GPUs, OpenCL and CUDA, which have proven to be high-performance technologies that solve a wide range of problems and allow achieving goals for developing the technological world.

Keywords: OpenCL and CUDA technologies , GPUs , information technology .

Chapter I:

1.1. Introduction:

The graphics processing unit (GPU) is an important part of any computer or smartphone that handles graphics. The GPU transforms digital data into colorful, moving images on the screen , the GPU is especially important in applications and games that require high-quality and smooth loading and playback of (3D) graphics. It transforms digital data into 3D images and applies effects, shading and lighting to graphics to achieve a realistic visual experience for the user , in addition, the GPU is used in other processes such as video encoding and decoding, image processing, and handling virtual reality and augmented reality technologies. It is also important in crypto currency mining operations, scientific calculations, and graphical analysis , the GPU improves graphics performance, speeds up renderings, and improves the user experience in many applications and games.
OpenCL and CUDA are parallel programming technologies that allow developers to speed up calculations using graphical processing units , by using these techniques, users can significantly improve the performance of computer applications, reduce execution time, and increase system efficiency in general.
OpenCL and CUDA are based on the same basic concept, which is to distribute the calculation to different processing devices to speed up its execution. However, they differ in some important ways.

OpenCL relies on a wide range of different hardware devices, while CUDA relies on NVIDIA technology found only in NVIDIA graphics cards. This means that OpenCL can run on a variety of devices, including central processing units (CPUs), graphics processing units (GPUs), field dividers (FPGAs), and others, while CUDA is limited to NVIDIA graphics cards only [1, 2] .
The languages used in both libraries are different. OpenCL can use programming in different languages such as C, C++, Python, etc., while CUDA uses C/C++ programming.
Our study aims to compare two software technologies on graphics processing units, OpenCL and CUDA, which have proven to be high-performance technologies that solve a number of problems and allow achieving the goals of creating an advanced technological world.

1.2. Research problem :

The research problem is to compare two software technologies on graphics processing units, OpenCL and CUDA, because these two technologies are considered high-performance technologies and allow the development of information technology and solve many problems.

1.3. Research importance:

The importance of research in comparing the two technologies is to create a technological world characterized by speed and high performance. Comparing software techniques also opens the way for technological development in the field of graphics processing, which has become of high importance in processing images, graphic programs, videos, games and many other tasks.

1.4. Research goals:

1. Support and development of OpenCL and CUDA GPU technologies.
2. Increasing the experience of programmers to identify software codes in order to develop them for high performance and speed.
3. Finding solutions to some potential negatives.

1.5. Research hypothesis:

The research hypothesis aims to develop and support the techniques of graphical units OpenCL and CUDA as they are fast characterized by high performance and speed and solve many problems and contribute to technological development.

Chapter II: Research methodology and Related work:

2.1. Research Methodology:

This research is based on programmatic analysis of OpenCL and CUDA graphics unit technologies:

2.2. Theoretical aspect and The practical aspect: In these aspects, the following are discussed:

On the theoretical side: Topics related to OpenCL and CUDA graphical unit technologies are studied.
On the practical side: OpenCL and CUDA will be programmatically analyzed to compare the two technologies for technological development, as graphic processor technologies are an important part of our lives and solve many problems.

2.3. Related work:

• **Weber et :** They compared the programming of multi-core graphics processors OpenCL and CUDA , they used applications to accelerate the performance of graphics units, and compared several applications for processing graphics units. Their study showed the possibility of application in OpenCL technology, but it showed a loss in performance [3].

• **Kamran et :** They worked on comparing the performance of OpenCL and CUDA graphics units, and their study showed that the two technologies use a complex and almost identical kernel, and that the way each technology works is that a specific algorithm or application is used to carry out each work [4] .

• **Memeti et :** They compared development through increased programming writing and performance on the one hand and on the other hand compared the energy consumption of OpenCL and Open MP , Open ACC technologies. Their study proved that programming OpenCL and CUDA technologies is more difficult than programming other technologies and in terms of energy consumption, OpenMP has equal performance and that its energy consumption in one scale OpenCL is better than OpenACC in another[5] .

Chapter III : Research Procedures

3.1. Theoretical Framework of Research:

OpenCL and CUDA technologies

CUDA is a platform that can significantly improve computing performance through the use of Nvidia GPUs. It is a C-like programming language with a compiler and computing libraries on the GPU. CUDA uses a large number of separate threads for calculations. All of them are grouped into a hierarchy - grid / block / thread. Grid is the top level in the model - it corresponds to the kernel and unites all the threads that this kernel executes. Grid is a one-dimensional or two-dimensional array of blocks. Each block block is a completely independent set of coordinated threads. Each thread has its own memory (local), inaccessible to other threads. Within a block, threads have shared memory and synchronous execution. Threads from different blocks cannot communicate, but all threads have access to a common global memory. A CUDA program (as in OpenCL) is divided into two parts: the first part is control, the second is computational. The central processing unit (CPU) acts as the control device (host), and the GPU is used as the computing device (device). On the central processor (host) only sequential parts of the program algorithm are executed, preparing and copying data to the device, setting parameters for the kernel and launching it. Parallel parts of the algorithm are formed into kernels, which are executed on a large number of threads on the device [6,7].

OpenCL is an open API (English application programming interface) — a description of the ways (a set of classes, procedures, functions, structures, or constants) in which one computer program can interact with another program for parallel computing, designed so that GPUs and other coprocessors can work in tandem with the CPU to provide additional computing power[8].

Like CUDA, OpenCL uses a work-item hierarchy. Streams are grouped into work-groups. Each thread has its own memory (private), as well as access to the group's shared memory (local) and the device's global memory (global.

In an OpenCL program, a computing device is selected from a list of platforms and their devices. A GPU is usually used, but a CPU can also be selected as the device.

The overall structure of the program is more complex than a similar CUDA program, as it is necessary to perform additional steps to configure the runtime and prepare code for the device[9].

On the CPU, the first step is to obtain information about platforms, select devices, and prepare the context for them. The next step is to create a kernel: get and compile the program text for execution from the source code. To run the resulting kernel, it is necessary to allocate memory and prepare data for processing, create a queue of device commands and assign kernel execution parameters. Upon completion of all work, it is necessary to release resources: context, program queue, memory allocated for data [10].

3.2. Matrix multiplication task

The task is to find the matrix C [M*K] obtained by multiplying two given rectangular matrices A [M*N] and B [N * K] with real coefficients .

3.3. Kernel implementation on CUDA

The first implementation is based on global memory (the software implementation is shown in Figure.1).
The algorithm works as follows:
1. Matrices A and B are split into blocks of a given size.
2. Each block has flows, the number of which depends on the block size. The coordinates of each stream are determined by coordinates within the block.
3. Each stream reads one row from matrix A, one column from matrix B, and calculates the corresponding element of matrix C.

```
__global__ void matrix_multiply_kernel(double* a, double* b, double* c, int m, int n, int
k) {
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
double sum = 0.0;
if (col < k && row < m){
for (int i = 0; i < n; i++)
sum += a[row * n + i] * b[i * k + col];
c[row * k + col] = sum;
}
}
```

Figure 1. The first implementation is based on global memory .

The (global) keyword is the entry point to a function for executing code on a GPU device. The BlockIdx variable determines the position/index of the block within the grid, BlockDIM — the block size, and ThreadIdX — the flow index within the block.
It is generally accepted that global memory works quite slowly, therefore, in order to speed up calculations and streams, let's consider the second version of the matrix multiplication algorithm with “shared” memory.
The main features of the shared memory algorithm between threads (shown in Figure.2) are:
1- Matrices A and B are split into disjoint blocks that are loaded into shared memory, which is much faster to access than global memory.
2- To reduce the number of global memory accesses, each thread loads only a portion of matrices A and B into shared memory.
3- Each thread is responsible for calculating the sum of row products per column of sub matrices A and B and stores only one value in C.
4- When each stream has calculated the block sum, the next batch of blocks of sub matrices A and B is loaded.
5- The calculations are completed when all subblocks of matrices A and B have been processed.

```
global__ void matrix_multiply (double* C, double* A, double* B, int wA, int hA, int wB){
int bx = blockIdx.x; // Block coordinate along the X axis
int by = blockIdx.y; // Block coordinate along Y axis
int tx = threadIdx.x; // The flow coordinate within the block
int ty = threadIdx.y; // The flow coordinate within the block
int aBegin = wA * BLOCK_SIZE * by; // The first submatrix from A
int aEnd = aBegin + wA – 1; // The last submatrix from A
int aStep = BLOCK_SIZE; // step
int bBegin = BLOCK_SIZE * bx; // First submatrix from B
int bStep = BLOCK_SIZE * wB; // step
double Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
// Declaring variables to use shared memory
__shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];
// Each stream loads only 1 element in the block
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];
// Synchronize the execution of all threads
__syncthreads();
// Each thread calculates only one element of matrix C
for (int k = 0; k < BLOCK_SIZE; ++k) {
Csub += As[ty][k] * Bs[k][tx];
}
__syncthreads();//Waiting for all threads to complete
}
C[wB * BLOCK_SIZE * by + BLOCK_SIZE * bx + wB * ty + tx] = Csub; // result to global memory
}
```

Figure 2. the matrix multiplication algorithm with “shared” memory.

3.4. kernel implementation on OpenCL

The main difference between the implementation of the program on OpenCL and the implementation on CUDA is the directives used. The principle and approach to calculation remain the same.

The (kernel ID) is used to indicate the function that will be called on the host and executed on (global) device in CUDA. Using the get_group_id function, we get the work-group number (block in CUDA), and the work-item identifier (thread in CUDA), which is currently running, is obtained using (get_local_id) . The barrier function is used to synchronize the operation of all streams.

3.5.Gravity problem

The initial position of N point bodies. Each body is characterized by mass m and initial velocity v. It is assumed that no forces other than gravitational forces act on the bodies. Under the influence of gravity, bodies change their position as well as their speed. It is necessary to simulate the interaction of bodies with each other after a given period of time and determine the final coordinates.

The mathematical model of the problem uses the following formulas:

The law of universal gravitation:

$$\vec{F} = G\frac{m_1 \times m_2}{r^2}. \qquad \text{(1)}$$

2. The rule of adding forces: if several forces act on a body, they can be replaced by one force equal to the vector sum of all initial forces.

3. Newton's second law:

$$\vec{F} = m\vec{a}. \qquad \text{(2)}$$

The interaction of bodies is modeled step by step using discrete periods of time of fixed duration dt. (At) each step, the force acting on each body at the initial point in time is calculated, and the acceleration is determined from the force found.
For the sake of simplicity, it is assumed that all bodies are located in the same plane. The position of each body is determined by two coordinates (x,y) [11] .

3.6. Core implementation on CUDA

As a limitation, we will consider the gravitational constant to be 10, the time step is 0.1 seconds. For cases where the interacting bodies are close to each other, we enter the maximum force (Fmx) equal to 1, the main feature of the implementation of the gravitational problem is that the calculation of new coordinates and velocities of bodies is divided into two stages: 1. The calculation of the forces acting on the particle occurs on the GPU device. As a result, the received data is returned to the CPU. 2. Coordinates and velocities are calculated on the CPU based on the forces received from the GPU. We will consider the first implementation option shown in Figure.3 using the device's global memory. When calculating, each thread calculates the forces exerted by other particles on one specific particle[12,13].

```
__global__ void forceGpu(double4* p, double4* f, double* m) {
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < N){
for (int j = 0; j < N; j++){
if (i == j) continue;
double dx = p[j].x – p[i].x;
double dy = p[j].y – p[i].y;
double r_2 = 1 / (dx * dx + dy * dy);
double r_1 = sqrtf(r_2);
double fabs = gravity * m[i] * m[j] * r_2;
if (fabs > Fmax) fabs = Fmax;
f[i].x = f[i].x + fabs * dx * r_1;
f[i].y = f[i].y + fabs * dy * r_1;
}
}
}
```

Figure 3. the matrix multiplication algorithm with "shared" memory.

The number of the particle for which the current thread will calculate forces is determined based on the values: blockIdx – block position inside the grid, blockDim – block size, threadIdx – thread index inside the block. The second option for calculating forces is based on the use of shared memory by threads inside a block (shown in Figure.4). All bodies are divided into groups. Each thread calculates the total force exerted on the body by other bodies within the block until all blocks have been processed.

```
__global__ void forceGpu(double4 p, double4 f, double* m) {
int i = blockDim.x * blockIdx.x + threadIdx.x;
//  Variables in device shared memory
_shared_ double sm[BLOCK_SIZE]; //  particle masses
_shared_ double4 spos[BLOCK_SIZE]; // particle coordinates

for (int tile = 0; tile < gridDim.x; tile++) {
//  each stream receives masses and coordinates for 1 particle
spos[threadIdx.x] = p[tile * blockDim.x + threadIdx.x];
sm[threadIdx.x] = m[tile * blockDim.x + threadIdx.x];
__syncthreads();//Synchronizing execution of all threads
for (int j = 0; j < BLOCK_SIZE; j++) {
int gj = tile * BLOCK_SIZE + j;
if (i == gj || gj >= N || i >= N){ continue; }
... //  calculation dx, dy, r_2, r_1, fx, fy
}
__syncthreads();//Waiting to move to a new block}
}
```

Figure 4. use of shared memory by threads inside a block.

3.7.Kernel implementation on OpenCL

As in the case of the matrix multiplication problem, the implementation of the two program variants on OpenCL is completely identical to that on CUDA, except for the constructs and directives used. Special functions are used to get the current body number: get_local_size to determine the size of the workgroup, get_group_id to determine the group number, and get_local_id to determine the number of the work item in the group. In the implementation of a gravity task with shared memory within a workgroup, the __local directive is used to define the local memory of the group and the barrier() function is used to synchronize threads within the group to continue processing the next block.

3.8.Comparison of work results

As a result of a series of tests with a constantly increasing matrix size, OpenCL and CUDA technologies showed almost identical results in implementations with global memory. The shared memory option on CUDA gave a slightly better result in terms of time shown in Figure.5.

In the gravitational problem, the calculation results showed a small scatter. The global memory implementation using OpenCL technology was slightly slower than CUDA. The shared memory implementation on OpenCL showed better results than on CUDA shown in Figure.6.

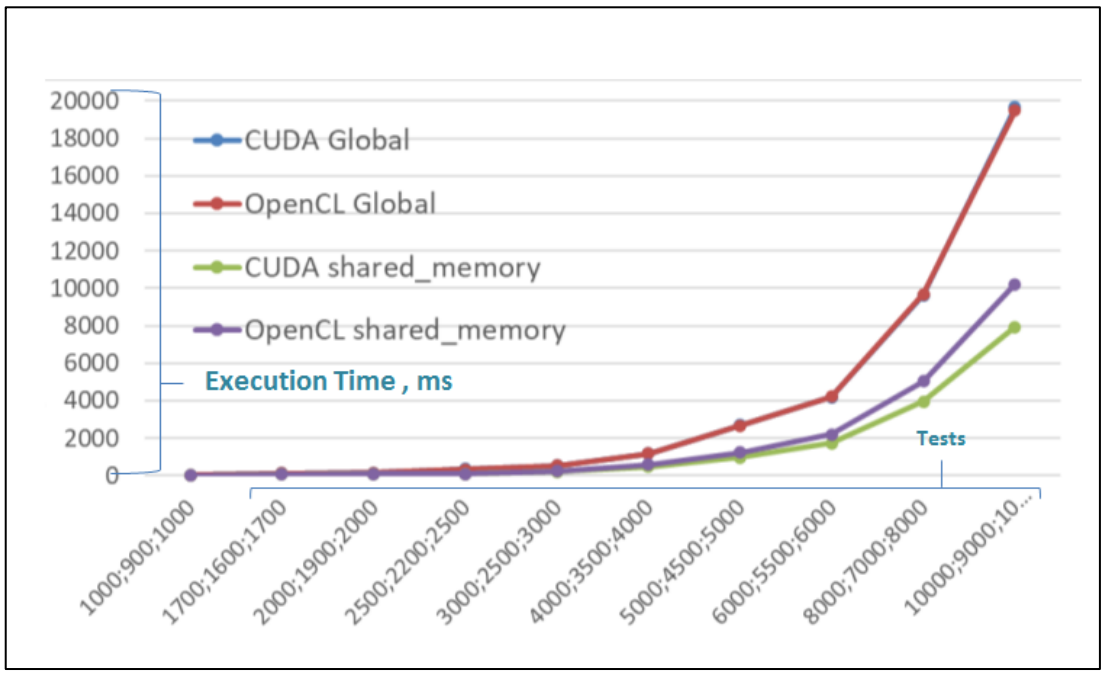

Figure 5. Running time of matrix multiplication programs

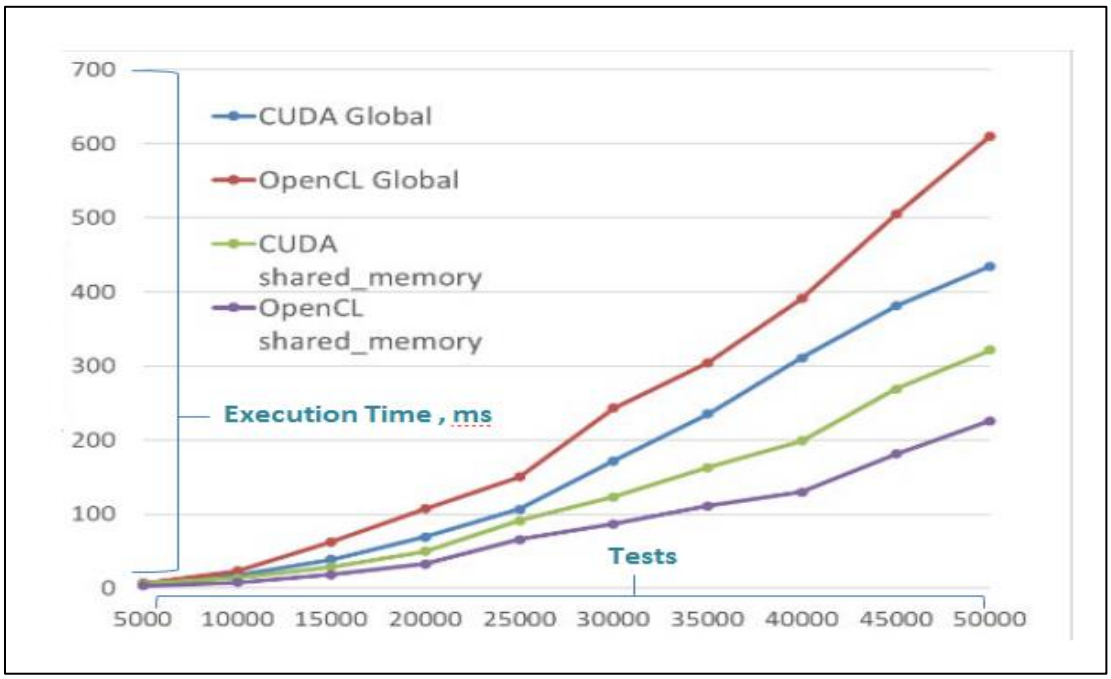

Figure 6. Running time of programs for solving the gravitational problem.

Chapter IV: Conclusions and future work:

4.1Conclusions:

Conclusion

This article is devoted to a comparison of popular GPU programming technologies CUDA and OpenCL. During the work, the program code for the tasks under consideration was implemented and optimized to obtain the best result. Launch cycles were carried out and metrics were collected and presented on the graph. As a result, we can conclude that these technologies are very similar in approach and organization and, based on the results of the tests, showed comparable results on the tasks under study.

4.2 future work:

In light of the results reached by the researcher comparing the GPU programming techniques, OpenCL technology and CUDA technology, the researcher recommends periodically conducting improvements and analyzes of the GPU programming technology, OpenCL technology and CUDA technology, to obtain the best results to keep pace with the scientific and technological era.

References

[1] Kirk, D. (2007, October). NVIDIA CUDA software and GPU parallel computing architecture. In ISMM (Vol. 7, pp. 103-104).

[2] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., & Dongarra, J. (2012). From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing, 38(8), 391-407.

[3] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL:Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing, 38(8):391–407, 2012.

[4] K. Karimi, N. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. Technical report, D-Wave Systems Inc., 2011.

[5] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking OpenCL, OpenACC, OpenMP,and CUDA: programming productivity, performance, and energy consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, pages 1–6. ACM, 2017.

[6] Asaduzzaman, A., Trent, A., Osborne, S., Aldershof, C., & Sibai, F. N. (2021, April). Impact of CUDA and OpenCL on parallel and distributed computing. In 2021 8th International Conference on Electrical and Electronics Engineering (ICEEE) (pp. 238-242). IEEE.

[7] Breyer, M., Van Craen, A., & Pflüger, D. (2022, May). A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware. In International Workshop on OpenCL (pp. 1-12).

[8] Nishitsuji, T. (2023). Basics of OpenCL. In Hardware Acceleration of Computational Holography (pp. 83-95). Singapore: Springer Nature Singapore.

[9] Wang, Z., He, B., Zhang, W., & Jiang, S. (2016, March). A performance analysis framework for optimizing OpenCL applications on FPGAs. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 114-125). IEEE.

[10] Kirk, D. and Hwu, W., Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers, 2010.

[11] Parallel methods for solving the n-body gravitational problem [Electronic resource]: database.(http://edu.mmcs.sfedu.ru/pluginfile.php/5634/mod_resource/content/5/OMPMPIGravit2019.pdf/).

[12] Demidov, D., Ahnert, K., Rupp, K., & Gottschling, P. (2013). Programming CUDA and OpenCL: A case study using modern C++ libraries. SIAM Journal on Scientific Computing, 35(5), C453-C472.

[13] Fang, J., Varbanescu, A. L., & Sips, H. (2011, September). A comprehensive performance comparison of CUDA and OpenCL. In 2011 International Conference on Parallel Processing (pp. 216-225). IEEE.