Parallel Acceleration on Manycore Systems and Its Performance Analysis: OpenCL Case Study

Rafael Alejandro Vejarano, Phuong Thi Yen and Jeong-Gun Lee

Dept. of Computer Engineering, Hallym University, South Korea
rafaelvejarano@hallym.ac.kr

Abstract

OpenCL (Open Computing Language) is a heterogeneous programming framework for developing applications that executes across a range of device types made by different vendors[11] which efficiently maps to both heterogeneous and homogeneous, single or multiple device system consisting of CPUs, GPUs and others types of devices. OpenCL provides many benefits in the field of high-performance computing and one of the most important aspects is its portability. This paper presents a comparison of the performance of OpenCL executing a matrix multiplication over a manycore CPU and GPU with performance analysis. The analysis are carried out to understand manycore CPU and GPU performance characteristics. Such analysis approach can be further extended to include more system parameters and refined to fit the actual execution time of parallelized applications. The simulation uses Ubuntu 12.04 in a desktop with an Intel i7 960 processor and a graphic card Nvidia GeForce GTX 460.

Keywords: *OpenCL, Matrix Multiplication, Kernel, CPU/GPU Codesign*

1. Introduction

Nowadays, processors with two or more cores are common in notebooks, tablets and smart phones, delivering additional performance. However, parallel software developers must contend with problems not encountered during sequential program development [1]. Parallel Computing lets us solve computational and data-intensive problems using manycore processors. Currently, GPUs (Graphics Processing Units) are used not only for graphics applications, but also for non-graphics applications, so-called GPU computing or GPGPU (General-Purpose computation on GPUs) [2]. Due to their high floating-point operation rates and memory bandwidths, GPUs can accelerate various science and engineering computations. The most popular development tool for scientific GPU computing has proved to be CUDA (Compute Unified Device Architecture) [3], provided by the manufacturer NVIDIA for its GPU products. However, CUDA is not designed for heterogeneous systems, while OpenCL programming model, created by the Khronos Group, supports cross-platform, parallel programming of heterogeneous processing systems.

The main goal of this paper is to investigate OpenCL as a programming standard for parallel processing architectures and provide some analytical background of the parallelized application. The analytical model can be used as a decision equation for mapping applications to two different computing resources (CPU and GPU). The matrix multiplication solution can be executed over a GPU or CPU without changing the code, reducing enough the execution time and getting a highest rate of float point per seconds compared with the serial program. A matrix multiplication program is implemented using OpenCL language over CPU and GPU devices. Performances between serial and parallel versions of different devices are compared.

The paper is organized as follows: A short introduction on OpenCL is presented in Section 2, Section 3 shows OpenCL architecture with its standard models, Experiment results and performance analysis are demonstrated in Section 4 and 5. Lastly, in Section 6, we conclude the paper.

2. OpenCL

OpenCL is a standard for programming heterogeneous computers built from CPUs, GPUs and other processors. It includes a framework to define the platform in terms of a host (e.g. a CPU) and one or more compute devices (*e.g.*, a GPU) plus a C-based programming language for writing programs for the compute devices [4]. Using OpenCL, a programmer can write task-based and data-parallel programs that use all the resources of the heterogeneous computer [2]. OpenCL exploits the massive-parallelism of modern processors, embedded devices and graphics processors (GPUs) providing many properties such as high-performance computing and portability. OpenCL-coded routines, called kernels, can execute on GPUs and CPUs from such popular manufacturers as Intel, AMD, Nvidia, IBM and much more [11]. It provides a top level abstraction for low level hardware routines as well as consistent memory and execution models for dealing with massively-parallel code execution [6].

3. The OpenCL architecture

One of OpenCL's strengths is that this architecture does not specify exactly what hardware constitutes a compute device. Thus, a compute device may be a GPU, or a CPU. Devices contain one or more compute units (processor cores). These units are themselves composed of one or more single-instruction multiple-data (SIMD) processing elements (PE) that execute instructions in lock-step [7].

3.1. The platform model

The platform model defines the relationship between the host and one or more devices. The host sets ups a kernel for device to run and instantiates it with some specified degree of parallelism [8]. The OpenCL platform model is depicted in Figure 1. A host (usually a CPU) is connected to one or more OpenCL compute devices (*i.e.*, a GPU or a CPU) which contain an array of functionally independent compute units. Compute units are further divided into processing elements.

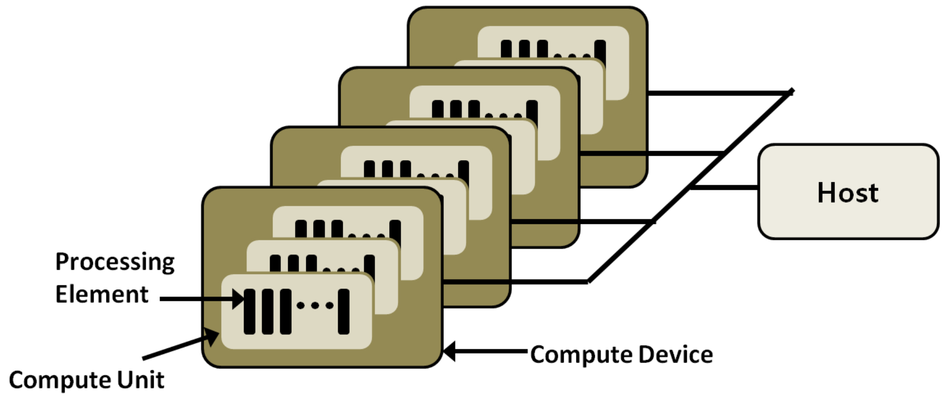


Figure 1. OpenCL Platform Model

3.2. The Execution Model

Execution process of an OpenCL program is divided into two parts: a host program executed on the host and kernels executed on one or more OpenCL devices. The host program defines a context for the kernels and manages their execution [5]. The context coordinates the mechanisms for host-devices interaction, manages memory objects and keeps tracks of the programs and kernels that are created for each device. In the execution model, context, command queue and buffer are created to allow command and data passage from the host to the devices and vice versa.

3.3. The Memory Model

OpenCL defines four memory spaces: private, local, global and constant. Private memory is memory that can only be used by a single processing element (work-item). This is similar to register in a single compute unit or a single CPU core. Local memory is memory that can be used by all work-items in a work-group (compute unit). Global memory is visible to all compute units on the device. Constant memory is a memory that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host processor is responsible for allocating and initializing the memory objects that reside in this memory space.

3.4. The Programming Model

The OpenCL execution model supports *data parallelism*, *task parallelism*, as well as supporting hybrids of these two models. The primary model driving the design of OpenCL is data parallel [7]. OpenCL provides a common language, programming interfaces, and hardware. OpenCL devices may or may not share memory with the host CPU, and typically have a different machine instruction set, so the OpenCL programming interfaces assume heterogeneity between the host and all attached devices.

4. Experiments

In the experiment, matrix multiplication with varying size was used to test the OpenCL model. Matrix multiplication is a clear example for data-parallel concurrency optimizations since output data range consists of a set of independent computations tasks [5]. Figure 2 shows the sequential solution which is a simple triple-for loop with two outer loops interactive over the row and column index and the inner for loop performing a dot-product of the row and column vectors.

```
void MatrixMul_seq(int hA, int wA, int wB, float *A, float *B, float *C)
{
    int i, j, k;
    for (i=0; i<hA; i++)   {
        for (j=0; j<wB; j++)   {
            for(k=0; k<wA; k++) {
                C[i*hA+j] += A[i*hA+k] * B[k*wA+j];
    }   }   }
}
```

Figure 2. Matrix multiplication sequential code

Figure 3 shows the kernel parallel code written in OpenCL. Size of the matrices are represented by variables hA, wA, wB which are set to be the same to make quadratic matrices. Both matrices are populated randomly. The OpenCl function get_global_id( ) indicates the work item ID which is supposed to compute the output which has the corresponding location ID in matrix C.

The kernel takes as parameter 3 different matrices A, B are input and C, the output , where the result will be store. heighA, heighB, widthA, widthB corresponds to height and width of the matrix A and width of matrix B and are equal to a constant value. First, global indexes are obtained using the OpenCL function get_global_id(); those indicate the initial position of matrix A and B. Then the partial sum of the multiplication is performed in a unique “for” loop of the kernel and then copy on the device the result of the multiplication. Both matrices are populated randomly.

```
__kernel void vecadd( __global float *A, __global float *B,
                                    __global float *C, int hA, int wA, int wB)
{
  // Get the work-item’s unique ID
  int row = get_global_id(1);
  int col = get_global_id(0);
  int sum=0;

   // Assign C[i,j] calculation to a thread
   for(int i = 0; i < wA;i++)
      sum+= A[row*wA+i] * B[i*wB + col];

   C[row*wB+col]=sum;
}
```

Figure 3. OpenCL Kernel code

Lastly, to measure time executed in the program, timing event was created. OpenCL function clGetEventProfilingInfo ( ) can be used to do this task.

The devices used for the experiments are listed in Table 1. OpenCL requires an environmental setup on the host before launching kernels to run, according to Peng Du [9], setting up the kernel takes longer than the kernel execution itself. Peng Du observed that compiling OpenCL source code takes most time in initialization.

Table 1. Devices characteristics

Vendor	Type	# of Cores	Clock (MHz)	Global Memory (MB)	Local Memory (KB)	Max. Work-Item Size	Max. Work Group Size
NVIDIA GTX 460	GPU	**336**	1600	1024	48	1024x1024x64	1024
INTEL Core i7	CPU	**8**	3200	2047	32	1024x1024x1024	1024

Creating an OpenCL program requires writing codes for the host side, as well as for the device side. An OpenCl program needs special configuration in order to execute the kernel and retrieve results. Table 2 is a step-by-step guide to implement OpenCl.

Table 2. OpenCl Step Implementation

STEP	Main Function
1. Discover and initialize the platforms	clGetPlatformIDs ()
2. Discover and initialize the devices	clGeDeviceIDs ()
3. Create a context	clCreateContext()
4. Create a command queue	clCreateCommandQueue()
5. Create device buffers	clCreateBuffer()
6. Write host data to device buffers	clEnqueueWriteBuffer()
7. Create and compile the program	clBuildProgram()
8. Create the kernel	clCreateKernel()
9. Set the kernel arguments	clSetKernelArg()
10. Configure the work-item structure	
11. Enqueue the kernel for execution	clEnqueueNDRangeKernel()
12. Read the output buffer back to the host	clEnqueueReadBuffer()
13. Release OpenCL resources	

5. Experiment Results

For sequential runs, the matrix size starts with data elements randomly generated and only one core was fully occupied. The size was varied from 1000x1000 until 6000x6000. While increasing the size of the array, the serial program execution time considerably increases. Execution times in serial and parallel versions over CPU and GPU devices are presented in Table 3. Note the times are only for kernel executions.

Table 3. Execution times and speedup over CPU and GPU devices

Matrix Size	T_{serial} (s) (serial version)	T_{CPU} (s) (parallel version)	Speedup T_{serial}/T_{CPU}	T_{GPU} (s) (parallel version)	Speedup T_{serial}/T_{GPU}
1000	11.283	0.774	14.68	0.156	72.33
2000	92.188	4.718	19.54	0.548	168.23
3000	405.846	18.831	21.55	1.411	287.63
4000	967.419	42.519	22.75	3.222	300.25
5000	2,058.774	98.305	20.94	6.083	338.45
6000	3535.899	166.963	21.18	10.499	336.78

As expected, parallel implementation outperforms the serial one and powerful 336-core GPU device outperforms 8-core CPU device. Note that we observe *super-linear* speedup when running program on CPU device. Intel Core i7 has 8 cores with one host core and 7 device cores. The factor of super-linearity or sub –linearity can be evaluated by "*efficiency*" that is given by the following equation.

$$Efficiency = {}^{Speedup}\big/_{\#\_of\_cores} \quad (1)$$

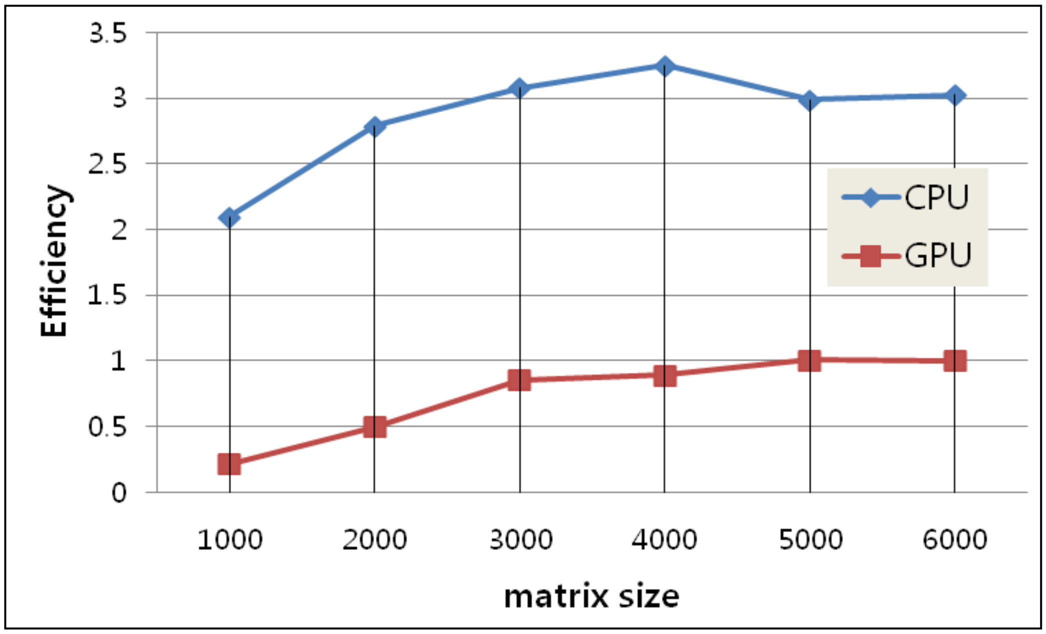


Figure 4. Speedup efficiency between CPU and GPU

Figure 4 shows the efficiency we achieved from parallel execution of matrix multiplication on manycore CPU and GPU. As shown in the figure, high efficiency of the parallel execution was observed in manycore CPU.

5.1. Interpretation of manycore performance

According to Amdahl's law, the speedup can be achieved not more than the number of cores running in parallel (which is '7' in this case). *Super-linear* speedup happens when the task is greater than the cache size when executed sequentially, but can fit nicely in each accessible cache when executing in parallel. So-called cache effect results from the different memory hierarchies because in parallel computing, not only the number of processors changes, but also the size of caches from different processors, with the larger accumulated cache size. More or even all of the working sets can fit into caches, when the CPU requests the memory line, data is already in the cache and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation. It gets a pipeline effect that is only limited by the bandwidth of the memory bus [10].

Figure 5 shows speedup performance characteristics of matrix multiplications on a manycore CPU.

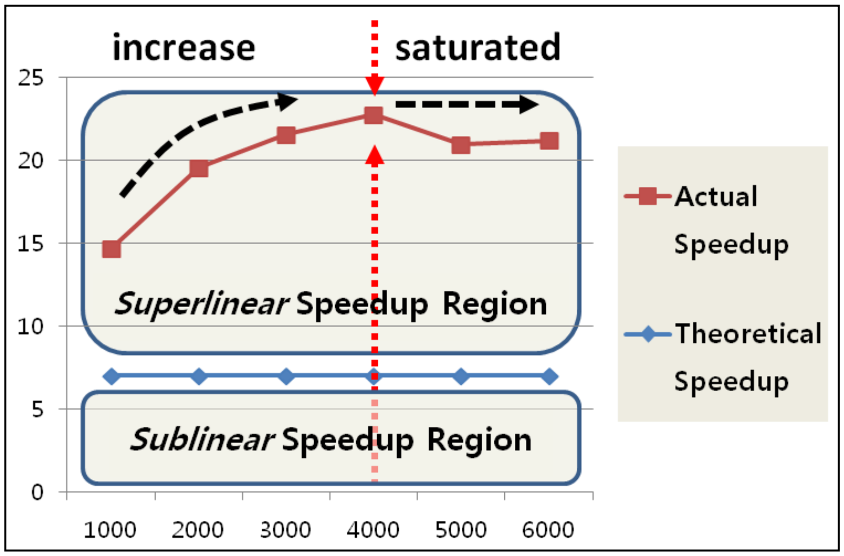


Figure 5. Speedup performance of matrix multiplications on a manycore CPU

5.2. Decision of computing resource type

If we can choose a best computing resource between CPU and GPU for a given workload application, then much efficient utilization can be possible by mapping an application to its best working computing platform. This sort of work are very popular in a "***hardware-software co-design***" which is a procedure for finding an optimal mapping of partitioned functionalities to CPU or FPGA/ASIC. Now, the hardware-software co-design principle can be evolved to "***CPU-GPU co-design***" in a modern heterogeneous computing era based on GPGPU.

The decision can be possible if we have pre-estimated working characteristics of a given application. Such a decision will depend on parameters such as; inherent parallelism in applications, overhead for the parallel computation such as memory copy between host and GPU device, and the number of parallel working cores, etc.

For an example, if a matrix multiplication is supposed to be executed then we have to decide whether host or GPU device runs it. If we have pre-estimated data such as T_{CPU} and T_{GPU} for a given matrix size together with overheads T_{CPU_Over} and T_{GPU_Over} for CPU and GPU executions respectively, then we can make proper decision for best performance. The decision equation will looks like the following equation.

$$\textbf{Decision Eq. : if(} T_{CPU} + T_{CPU\_Over} > T_{GPU} + T_{GPU\_Over} \textbf{) then Run the code on GPU else Run the code on CPU} \quad (2)$$

In general, T_{CPU_Over} is relatively smaller than delay parameters, so CPU-side execution time can be approximated by T_{CPU}. $TGPU_{\_Over}$ is a data-size depending parameters and non-linear to the amount of data to be copied between host and device. Some of GPU initialization time will be added to T_{GPU_Over}. For a GTX 460 GPU device, memory copy bandwidth is around 1.8GB/s, so the approximated value of T_{GPU_Over} can be calculated as the following equation.

$$T_{GPU\_Over}\text{(matrix-size)}= \{[\ 3\times 4B\times(\text{matrix-size})^2\ ] / 1.8GB\} + \text{CPU\_Initialize\_Time} \quad (3)$$

For the matrix multiplication, due to the high parallelism, always GPU outperform CPU for the case that matrix size is larger than 1000. When the matrix size reduces less than 100, then CPU is expected to outperform GPU due to overhead of GPU execution. In general, T_{CPU_Over} is relatively smaller than delay parameters, so CPU-side execution time can be approximated by T_{CPU}.

6. Conclusion

OpenCL was born for portability; it is good at cross-platform. But, the performance and programmability issues are becoming gradually more severe. Because OpenCL provides us with more details on architectures, it is quite difficult to program in OpenCL than in high-level CUDA. In order to generate efficient code in GPU, developers must understand the API. The fact that OpenCL works in a wide variety of platforms and hardware (CPUs and GPUs of different brands and architectures) has contributed to its growing success. Some has been commenting that much work remains to be done especially in the area of compilation.

The goal behind the OpenCl standard is to provide functionality, enabling a single OpenCL application to run across a variety of hardware platforms. Taking full advantage of OpenCL, thorough understanding of host applications and kernels is needed. This paper has demonstrated the effectiveness of manycore CPU and GPUs in dealing with the parallelization of matrix-vector multiplication on a very basic level as well the portability of OpenCL through different platforms. Furthermore, some analysis have been carried out to understand manycore CPU and GPU performance characteristics. Such analysis approach can be further extended to include more system parameters and refined to fit the actual execution time of parallelized applications.

Finally the analytical models can be used as basic partitioning criteria in a CPU-GPU co-design environment.

Acknowledgements

This research was supported by Hallym University Research Fund, HRF-201209-025.

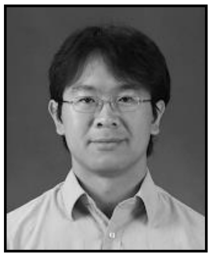
References

[1] A. Marowka, “A Study of the Usability of Multicore Threading Tools”, International Journal of Software Engineering and Its Applications, vol. 4, no. 3, **(2010)** July.
[2] P. P. Shete, P. P. K. Venkat, D. M. Sarode, M. Laghate and S. K. Bose, “Applying Object Oriented Design Patterns to CUDA based Pyramidal Image Blending - An Experience”, International Journal of Software Engineering and Its Applications, vol. 6, no. 2, **(2012)** April.
[3] NVIDIA CUDA, http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf.
[4] OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, John E. Stone, David Gohara, and Guochun Shi, Computing in Science & Engineering, vol. 12, no. 3, **(2010)**.
[5] June 2011 Cern Computing Seminar, http://indico.cern.ch/conferenceDisplay.py?confId= 138427.
[6] J. Thompson and K. Schlachte, “An Introduction to the OpenCL Programming Model”, Person Education, **(2012)**.
[7] The OpenCL Specification, http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.
[8] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry and D. Schaa, “Heterogeneous Computing with OpenCL”, **(2012)**.
[9] P. Dua, R. Webera, P. Luszczeka, S. Tomova, G. Petersona and J. Dongarraa, “From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming”.
[10] T. Kejser’s, "The Effect of CPU Caches and Memory Access Patterns, http://blog.kejser.org/2012/06/14/the-effect-of-cpu-caches-and-memory-access-patterns.
[11] Kronos Group, Conformant Products, http://www.khronos.org/conformance/adopters/conformant-products.

Authors

Rafael Alejandro Vejarano is a master student in Computer Engineering at Hallym University in South Korea. He got his bachelor's degree in at Technological University of Panama. His current research interests include networking, software for high-performance computer systems, parallel processing and e-learning.

Phuong Thi Yen is currently a master student in Computer Engineering at Hallym University in South Korea. She received her bachelor's degree of Electrical and Electronics Engineering in University Tenaga National in Malaysia in 2011. Her research interests are many-core microprocessor, system-on chip design and GPU-based parallel processing.



Jeong-Gun Lee is Associate Professor at Hallym University. He obtained his Ph.D degree in Information and communications from Gwangju Institute of Science and Technology, South Korea in 2005. His current research interests are Computer/Multi-Core Architecture, Asynchronous circuits, High-speed/low-power digital design, Application of asynchronous circuit designs to system on a chip, Network on Chip/Router Design, Arithmetic Circuits.