International Journal of Computer Applications (0975 – 8887)
Volume 82 – No3, November 2013

# A Methodology for Translating C-Programs to OpenCL

Krishnahari Thouti
Dept. of CSE
VNIT, Nagpur
India, pin-440010

S. R. Sathe
Dept., of CSE
VNIT, Nagpur
India, pin – 440010

## ABSTRACT

Graphics Processing Units (GPUs) is currently a common feature of high performance computing. Languages such as CUDA and Open Computing Language (OpenCL) are such programming models; provide a standard interface for achieving high performance across these GPU devices. However, because of the wide variety of architectural complexities of these GPU devices; often makes difficult to write programs for these platforms. One of the approaches to get rid off this difficulty is to parallelize sequential programs into equivalent parallel programs. In this paper, we present a methodology for parallelization of sequential C-programs with function calls to equivalent OpenCL programs with little assistance from programmer. Our proposed methodology identifies function calls and converts them into 'kernel' to be executed in parallel on GPU devices. To the best of our knowledge, there are no tools dedicated to conversion of C code to equivalent OpenCL code.

## General Terms

Parallel Computing, Parallel Algorithms

## Keywords

GPU Computing; OpenCL; Automatic Translation; Parallelization; Parallelizing Compilers

## 1. INTRODUCTION

Current High Performance Computing (HPC) platforms have multicore processors and Graphics Processing Units (GPUs) [1] as computing units. These GPUs have become a common feature of high performance computing. GPUs are highly parallel, multithreaded; many core processor with tremendous computational power and very high memory bandwidth. GPUs use aggressive multithreading so that whenever a thread is stalled, waiting for data, the thread can efficiently switch to execute another thread.

Achieving good performance on these devices requires explicit structuring of the applications to exploit parallelism. Parallel programming languages such as Brook+ [2], NVidia's CUDA [3], OpenCL (Open Computing Language) [4], have been recently introduced to help programmers in writing parallel programs to take benefit of GPUs for high performance computing. Still, even these new languages require the programmers to explicitly extract the parallelism from their applications. There are many factors concerning OpenCL and programmers have to create and manage thousands of threads, deal with concurrent execution, copy data between different processors, make use of scratchpad memories, and deal with issues such as memory access patterns, synchronization, race conditions and atomicity. It is, therefore, of great interest, for programmer's point of view, to develop support to facilitate automatic transformation of sequential programs into efficient parallel OpenCL programs.

However, there are few works are available on automatic parallelization for GPUs. Lee et al. [5] developed compiler framework for automatic translation from OpenMP to CUDA. The system handles both regular and irregular programs parallelized using OpenMP primitives. An automatic code transformation system that generates parallel CUDA code from input sequential C code is presented in [6]. In this, said system makes use of CLooG [7] generator that transforms a polyhedral representation of a program and affine scheduling constraints into concrete loop code and Pluto [8] optimizer that enables end-to-end parallelization. Another parallelization system that targets GPUs is that of Cornwall et al. [9] which perform source-to-source translations to help domain experts retarget an image processing library written in C++ to GPUs. CUDAlite [10] and hiCUDA [11] are source-to-source compilers which generate GPU code, but both require annotations in the original code. A CU2CL translator has been implemented in [12] for multi and many core architectures.

In this paper, we present a parallelization methodology to generate high performance GPU code from sequential C code. Our proposed methodology identifies function calls and converts them into 'kernel' to be executed in parallel on GPU devices. We focus heavily on the applicability of this translator in practice, and address its shortcomings and experimental results. To the best of our knowledge, there are no tools dedicated to automatic conversion of C code to OpenCL programs.

The rest of this paper is organized as follows. Section 2 introduces an overview of GPU architecture and OpenCL programming model and its issues are covered in Section 3. In Section 4, we propose our methodology C-to-OpenCL translation scheme. Next, experimental results and performance issues are provided in Section 5. Section 6 concludes with summary and future scope.

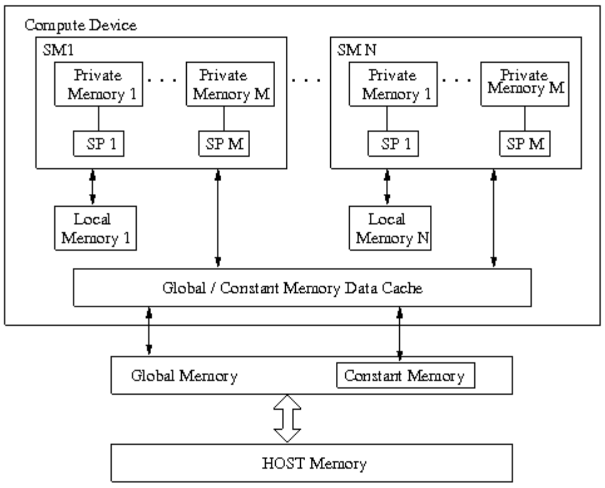

**Fig 1: Architecture of CUDA-enabled GPU Device**

## 2. GPU ARCHITECTURE

A GPU device is made up of hundreds of processing cores. As shown in Fig. 1, GPU is two-level architecture. It is made of vector processors at top level termed as Streaming Multiprocessors (SMs) and each SM contains eight processing cores, termed as, Symmetric Processors (SPs) grouped in an SIMD fashion. As a result, all SPs in an SM execute the same instruction. In this paper, we focus on CUDA-enabled NVidia Quadro FX 3800 GPU. It has 24 SMs, which makes for a total of 192 processing cores.

Each multi-processor is capable of creating, managing and executing concurrent threads with zero scheduling overhead. OpenCL uses a relaxed consistency memory model. The device memory is divided into global, local, and on-chip shared memory areas as illustrated in Fig. 1.

- Global memory is visible to all compute units on the device and accessible to all the threads. Whenever data is transferred from the host to the device, the data will reside in global memory. Any data that is to be transferred back from the device to the host will also reside in global memory.
- A 16KB Local memory is a scratchpad memory for a thread block and is shared amongst the threads running on multiple compute units. It is a very fast memory, as access time is very less.
- Constant memory is a part of global memory that stores variables whose values never change. It is a read-only memory.
- Private memory is memory that is unique to each thread. This is memory used within a work item that is similar to registers in a GPU multiprocessor or CPU core.

In OpenCL programs, data needed for computations on GPU is transferred from CPU to GPU through global memory and a number of threads are spawned. All the threads launched are independent of each other and their execution or ordering cannot be controlled by the user. Refer [3, 4, 17] for more details on this topic.

## 3. OPENCL PROGRAMMING MODEL AND ITS ISSUES

### 3.1 OpenCL Programming Model

An OpenCL program consists of two parts: one that executes on host (CPU) and other that executes on device (GPU). The function that executes on the GPU is called kernel. OpenCL is a restricted version of the C99 language with extensions appropriate for executing data-parallel code on a variety of heterogeneous devices. The OpenCL programming model is based on multi-threaded SIMT model. A typical OpenCL application consists of following steps, also shown in Fig. 2.

- Query to check whether OpenCL platform is present
- Get list of devices supported by OpenCL platform using '*clGetDeviceInfo()*'
- Create context from selected device then
    - Create one or more command-queues
    - Create programs to run on one or more GPU devices
    - Create kernel from these programs
    - Allocate buffer space on GPU devices
    - Write or copy data to and from host and GPU devices
    - Submit kernels to a command-queue for execution

A key concept of OpenCL programming model is the Work-Item. A work-item is a smallest execution entity. Every time a kernel is launched, a set of work-items, specified by programmer are launched, each one executing the same code. A set of work-items are organized in an N-dimensional grid, termed as work-group. Synchronization among work-items in the same work-group is achieved by using barriers. Work-items in different work-groups cannot synchronize with each other.

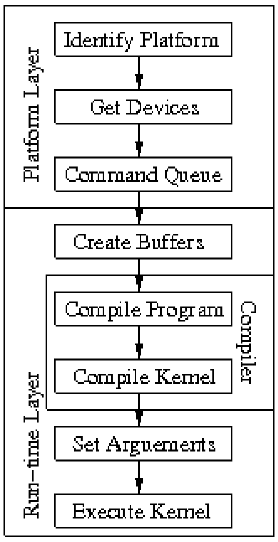

**Fig 2: Programming steps to write OpenCL application**

Fig. 3 shows OpenCL parallel execution model. The NDRange contains all work-items. In the Fig. 3, kernel is launched with a two-dimensional (2D) NDRange. All the work-items are given unique global index values. However, if they are organized in the form of work-groups, respective local index values are obtained from dimensional size of its work-group. The OpenCL API functions *get_global_size()* and *get_local_size()* cab be used to identify the dimensional sizes of NDRange on either x and y directions. Refer [13 - 17] for more details on this topic.

### 3.2 Issues Related to OpenCL

There are many factors concerning OpenCL; that is programmers have to explicitly create and manage thousands of threads, deal with concurrent execution, copy data between different processors, make use of different memory hierarchies of GPU devices, and deal with issues such as local work-group, global work-group parameters, barriers and synchronizations.

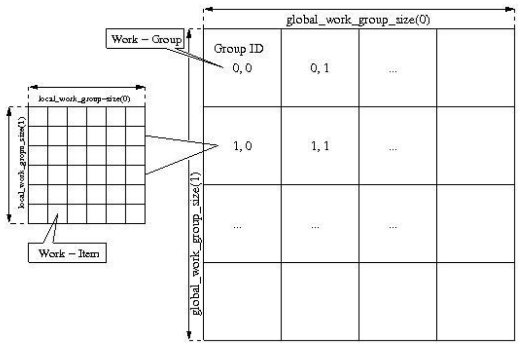

**Fig 3: OpenCL Parallel Execution Model**

It is, therefore, of great interest, for programmer's point of view, to develop support to facilitate automatic transformation of sequential programs into efficient parallel OpenCL programs covered in next section.

## 4. METHODOLOGY FOR C-to-OPENCL TRANSLATOR

In this section, we describe our proposed C-to-OpenCL translator, which translates C program to OpenCL based GPU program. We have presented a diagram of the various parts of translator in Figure 3. The translation consists of several steps: (1) Lexical Analysis & Parsing (2) Control and Array-flow Analysis (3) Dependence Analysis and transformations (4) GPU Buffer and Data Management (5) Syntactic post-processing and Parallel Program Generation. Fig. 4 shows schematic representation of these steps.

### 4.1 Phase 1: Lexical and Parsing

The input sequential C-program (strictly in C99 format) is fed into the first block which consists of a pre-processor phase, Lex Analyzer, and a Parser. In pre-processing phase, header files and some string substitutions are processed throughout the input code. This block checks whether the semantic constraints of the C-language, as per our format, have been properly satisfied.

A sequential C-99 program consists of “declaration-part” & “expression-part” as shown in Listing 1.1. The “declaration-part” consists of all variables and other data structures needed for program. The “declaration-part” is analyzed to get the array symbols and dimensionality of these array symbols. The “expression-part” contains “Function-Call” and other expressional statements. If “Function Definition” is identified as a candidate for parallel execution then “kernel code” is generated for executing on GPGPU device. The output of lexical analyzer is fed to parser.

The parser produces an Abstract Syntax Tree (AST), a symbol table and other useful information needed for dependence and inter-procedural analysis.

### 4.2 Phase 2 and 3: Control Flow and Dependence Analysis

The next stage of translator is analysis, which consists of several phases of analysis. The first step is to produce control-flow graph (CFG) by control-flow analysis. The CFG converts the different kinds of control transfer constructs in the program into a single form that is easier for compiler to manipulate. The next step is to construct Function Dependence Graph (FDG). The FDG is analyzed to determine possible candidates for parallelization. The loops and loop-nestings is the code to be parallelized.

```
#include <hdr_files>
void main()
{
   /* declaration–part */
   int i,j,k =1024;
   int a [1024] , b [1024] , c [1024];
   for (i=0; i<k; i++) {
   a[i]=b[i]=c[i]=i+3;
   }
   fro (a,b,c,k); /* Function - Call */
   print_result ();
}
```

Listing 1.1. Typical C-99 Program

Next Symbolic Array Dataflow Analysis is applied. Array dataflow analysis refers to computing the flow of values for array elements. With this information Pluto Compiler [8] will determine parallelism in loops. Array Dataflow information plays an important role for the automatic parallelization of sequential programs. The array information is used for allocating memory on GPU device and data to be transferred to the GPU device.

A typical example of a nested loop program is shown in Listing 1.2. It shows a for-loop with iterator “*i*”. The upper bound of the loop is specified by the variable “*size*”. Inside the for-loop, there is a statement which takes two input variables “*A*” and “*B*” on Right-hand side (Rhs) and produces one output variable “*Out*” on Left-hand side (Lhs variable).

```
void fro(int A[],int B[],int Out[],int size)
{
   int i ;
   for(i=0; i<size; i++)
   Out[i] = A[i]* B[i];
}
```

Listing 1.2. Function Block ‘fro’

Both the Rhs & Lhs variables in Listing 1.2 are enclosed by for-loop, spanned by iteration space. This space is mathematically described as a polytope $e(I; P)$ where $I$ represent the iterators (e.g. $i$) and $P$ represents the parameters (e.g. size). Two variables have dependency if and only if both access the same memory location. After analyzing the dependences, we apply Uni-Modular transformation technique [18] which takes dependence matrix as input and produces equivalent dependent-free code which can be run in parallel. So, in our case, “*fro*” is detected in the Function Detection Graph and control will be transferred to kernel module.

On analyzing the Function Block, our system will store following information in the symbol table shown in Table 1.

**Table 1. Read-Write Reference Table**

| | |
|---|---|
| A | Read Reference |
| B | Read Reference |
| Out | Write Reference |
| NDRange | Dimensionality (Iteration space) |

Let $R_f\ (R)$ = {set of all ‘read’ references of statement $S_k$ in Function Block $F_B$}

Let $R_f\ (W)$ = {set of all ‘write’ references of statement $S_k$ in Function Block $F_B$}

Hence, set of all data spaces accessed by all statements of Function Block $F_B$ will be

$$R_f = \sum_{\forall S_i \in S \text{ in } F_B} R_f\ (R) + R_f\ (W)$$

The NDRange accessed by the array references determines the buffer space needed on the kernel device.

### 4.3 Phase 4: GPU Buffer Management and Data Transformations

There are two ways to copy data from the host to the GPU compute device memory: (i) Implicitly by using “*clEnqueueMapBuffer*” and (ii) Explicitly through “*clEnqueueReadBuffer*”. Fig. 5 illustrates the standard dataflow between host (CPU) and GPU.

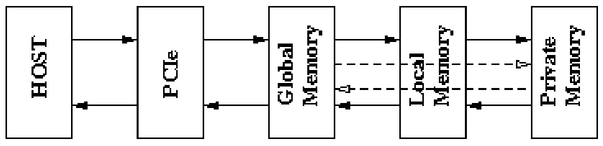

**Fig 5: Architecture of CUDA-enabled GPU Device**

In our translator, we consider only the second method of translating values between CPU and GPU. In OpenCL, data values that must be transferred from host to device are costly in terms of performance. There are two types of data movement (i) *kernel_in_stmt*: Data movement statements to move data from host to device. (ii) *kernel_out_stmt*: Data movement statements to move data from device to host.

The "read" and "write" modes of array symbols can be obtained from symbol table. The *kernel_in_stmt* will be sent to device using the "*clEnqueueWriteBuffer*" subroutine, defined in OpenCL specifications. Using Algorithm 1, CLooG [7] generates the code for data movement using data space polytopes. The algorithm to generate data movement in OpenCL code is outlined in Algorithm 1.

```
At Function Block $F_B$ do
{
   // $S$: Set of statements in $F_B$
   for each statement $s \in S$ do
         for each array $A$ do
                  find ND-Range
         end for
   define Polytope Boundary Region
   Apply Uni-Modular Transformation
   end for
   for each statement $s \in S$ do
         find kernel_in_stmt = "read" reference stmt
         find kernel_out_stmt = "write" reference stmt
   end for
 Output - Buffer space and Data Movement statements for GPU device
}
```

Algorithm 1 Generation of Data Movement Code

## 4.4 Phase 4: GPU Buffer Management and Data Transformations

The output from previous phases, gives parallel code which is further analyzed to give object code for OpenCL programming model. An OpenCL program consists of set of kernels that are identified as functions declared with the *__kernel* qualifier in the program source. The generation of OpenCL code involves declaration of OpenCL variables, an associated context, a program source or binary, successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log and the kernel object attached.

```
__kernel void fro(__global const int *A,__global const int
*B,__global const *Out, const int size)
 {
         int i ;
         i = get_global_id(0) ;
         Out[i] = A[i] * B[i];
 }
```

Listing 1.3. Generated GPU Kernel code for Listing 1.2

After OpenCL code is generated, the last step is to clean-up all the resources associated with OpenCL objects. This can be done by using release functions specified in OpenCL specifications. All the above phases require little assistance from the programmer wherever possible.

## 5. RESULTS AND ANALYSIS

To evaluate our framework, we investigated translations from sequential input program to output program tailored for parallel execution on GPU-equipped system. Translation is based on above mentioned translation scheme.

### 5.1 Experimental setup

All experiments were performed on 2.66 GHz Intel Xeon X 5650 (dual-core) systems with 4GB DDR3 main memory. The GPU device used in our experiment was NVidia Quadro FX 3800. The device has 192 processing cores with 1 GB 256 bit memory interface and memory bandwidth of 51.2 GB/sec. The GPU device was connected to CPU through X58 I/O Hub PCI Express. The environment used was Fedora operating system running Linux Kernel - 2.6.38.6-26.rc1 (FC15.x86_64). All experiments have been compiled with GCC 4.6.2 compilers. NVIDIA CUDA Toolkit 4.0 was used for GPU enabled experiments. We report application runtime (wall-time) for all test cases. We consider application runtime including runtime calls and GPU data transfers as an appropriate measure for our experiments. For test case, mean values from thirty runs are reported.

### 5.2 Experimental Results

An OpenCL program consists of two parts, host code (.c file) and a kernel code (.cl file). Our translation scheme creates two such files. For the above Listing 1.2, the kernel file generated is shown in Listing 1.3.

Table 2 shows the execution time of above said Program Listings 1.1 on the CPU and GPU. We took very large data of for experimentation. It can be seen from Fig. 6 that there is huge performance improvement, up to 14 times when using the GPU as compared to CPU.

## 6. CONCLUSION AND FUTURE SCOPE

The methodology we described can be considered a premature stage of what could become a framework devoted to parallelizing complier for OpenCL programming model. The problem of achieving correct and efficient parallel programs is made difficult by the various issues involved with OpenCL programming model including local and global work-group, work-items and other complex parallel architectural communications.

Our methodology can be extended to overcome above problems by a combination of good programming practice and using appropriate tools such as debuggers, profilers and performance analyzers. However, our translation currently has some limitations - (i) Input C-program should strictly follow C-99 format. (ii) Input program must have a "Function call". (iii) Global and local work-group size parameters are set to default values depending on the GPU device. In the near future, we plan to complete the development, testing and the evaluation of our translation scheme by using more scientific algorithms and create a publicly available release of the transformation software.

**Table 2. Execution Time calculation for CPU and GPU Device (in msec)**

| Vector size (in MB) | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 72 |
|---|---|---|---|---|---|---|---|---|
| **Seq (CPU)** | 4.68 | 9.42 | 18.81 | 37.6 | 75.18 | 150.54 | 300.63 | 338.18 |
| **OpenCL (GPU)** | 0.35 | 0.68 | 1.33 | 2.66 | 5.29 | 10.56 | 21.08 | 23.71 |

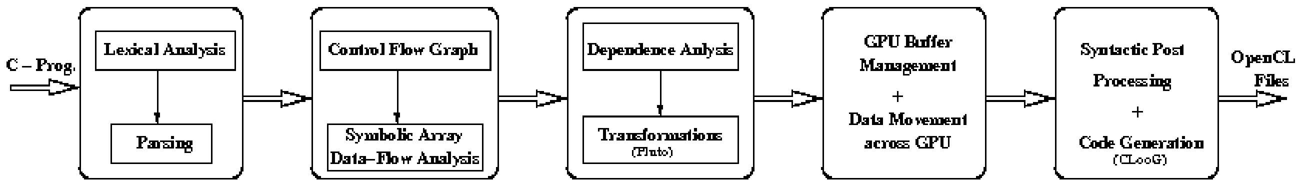
C – Prog.
Lexical Analysis
Parsing
Control Flow Graph
Symbolic Array Data–Flow Analysis
Dependence Anlysis
Transformations (Pluto)
GPU Buffer Management + Data Movement across GPU
Syntactic Post Processing + Code Generation (CLooG)
OpenCL Files

**Fig 4: C-to-OpenCL Code Translator**

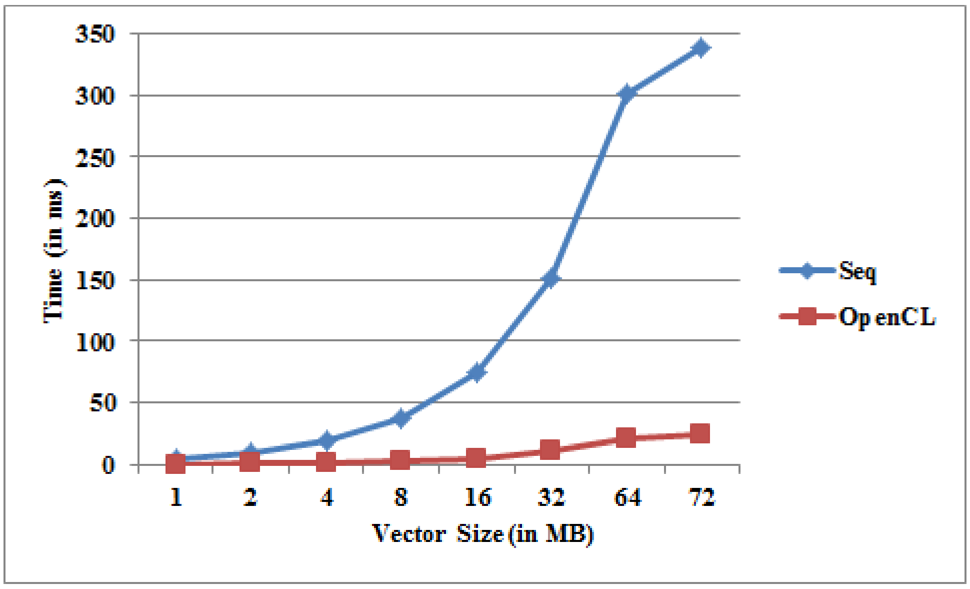

**Fig 6: Performance comparison of CPU v/s GPU**

## 7. REFERENCES

[1] General-purpose computations using Graphics hardware, http://www.gpgpu.org/

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fathalian, M. Houston and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," ACM Trans.Graph, Vol.23, No.3, 2004, pp. 777-786

[3] NVIDIA CUDA, http://developer.nvidia.com/cuda/

[4] OpenCL, http://www.khronos.org/registry/cl/

[5] Lee, S., Min, S-J., Eigermann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: PPoPP, pp. 101-110 (2009)

[6] Baskaran, M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA generation for Affine Programs. Compiler Construction. Lecture Notes in Computer Science, Vol. 6011. pp. 244-263 (2010)

[7] CLooG, The Chunky Loop Generator, http://www.cloog.org/

[8] Pluto, http://pluto-compiler.sourceforge.net/

[9] Cornwall, J.L.T., Beckmann, O., Kelly, P.H..: Automatically translating general purpose C++ image processing library for GPUs. In: POHLL. pp. 381 (2006)

[10] Ueng, S-z., Lathara, M., Baghsorkhi, S., Hwu, W-m.: CUDA-Lite: Reducing GPU Programming Complexity, Languages & compilers for parallel computing. Lecture Notes in Computer Science, Vol. 5335. pp. 1-15 (2008)

[11] Han, T.D., Abdelrahman, T. S.: hiCUDA: A high-level directive based language for GPU programming. In: GPGPU -2. pp. 52-61 (2009)

[12] Martinex, G., Gardener, M., Feng, W-c.: CU2CL: A CUDA-to-OpenCL translator for Multi-and-many-core Architectures. In: IEEE ICPADS. pp. 300-307 (2011)

[13] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, "Heterogeneous Computing with OpenCL", Morgan Kaufmann Publishers, 2011.

[14] AMD Accelerated Parallel Processing OpenCL Programming Guide, Advanced Micro Devices, Inc. 2012. http://developer.amd.com/appsdk

[15] A. Munshi, B. Gaster, T. Mattson, J. Fung and D. Ginsburg, OpenCL Programming Guide, Addison-Wesley Publishers, 2011.

[16] M. Scarpino, "OpenCL in Action," Manning publications, 2011.

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

[18] Banerjee, U.: An introduction to a formal theory of dependence analysis. In: Journal of Supercomputing. Vol.2, No.2 pp. 133-149 (1988).