OCLS: A Simplified High-Level Abstraction Based Framework for Heterogeneous Systems

Shusen Wu, Xiaoshe Dong, Heng Chen and Bochao Dang

Abstract In contrast with the increasing popularity of heterogeneous systems, programming on these systems remains complex and time-consuming. Developers have to access heterogeneous processors through explicitly and error-prone operations provided by low-level approaches like OpenCL. We present OCLS (OpenCL Simplified), a high-level abstraction based framework and its implementation as a minimal library on the top of OpenCL. OCLS shields hardware details, simplifies the development process and handles the environment configuration and data movement implicitly. Its APIs act like ordinary functions and require little prior training. OCLS thus reduces heterogeneous programming effort and relieves the programmers of low-level programming. We evaluated OCLS across a set of different benchmarks. The size of benchmarks rewritten in OCLS reduced by an average ratio of 35.4 %. In the experiment on both GPU and Intel MIC platforms with data sets in different size, OCLS yielded better performance than original OpenCL programs and showed a good stability and portability.

Keywords Heterogeneous programming · OpenCL · OCLS · Abstraction

1 Introduction

Heterogeneous systems employing different kinds of accelerators/co-processors are continuously dominating the high performance computing area according to the Top500 list [1]. With a 33.86PFlop/s peak performance, the Tianhe-2 supercomputer which uses Intel Xeon CPUs and Xeon Phi co-processors is currently the

S. Wu (✉) · X. Dong · H. Chen · B. Dang
School of the Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, Shaanxi, P. R. China
e-mail: wuss153@stu.xjtu.edu.cn

J.J.(Jong Hyuk) Park et al. (eds.), *Advances in Parallel and Distributed Computing and Ubiquitous Services*, Lecture Notes in Electrical Engineering 368,
DOI 10.1007/978-981-10-0068-3_7

fastest system worldwide. The cost-effectiveness, power-efficient and high performance accelerators/co-processors have caused the shift from homogeneous programming to heterogeneous programming. However, the state-of-art heterogeneous programming methods [2, 3] could be disappointing.

The current de facto standard for heterogeneous computing is OpenCL [4]. With vendor provided runtime support, OpenCL is available on a wide range of architectures. The theme of OpenCL programming is to offload parallel computations (kernels) on devices. To archive that, applications have to start with device query and environment configuration. Resource allocation on device, data transfer from host to device and vice verse after the kernel execution are also needed. All the operations above including kernel launch are done explicitly through the OpenCL APIs. Programmers deal with hardware details which may limit the portability of the program. This leads to tedious and error-prone code and makes heterogeneous programming complex and difficult.

We present a high-level abstraction based framework called OCLS to simplify the programming on heterogeneous systems. OCLS framework provides a single virtual processor abstraction for all heterogeneous systems and hides the hardware details for programmers through a library. The OCLS library encapsulates the OpenCL APIs and realizes automatic environment configuration. Data movements are handled at runtime implicitly with the help of a kernel data type defined in OCLS. The OCLS framework enables programmers to carry out heterogeneous computing in an ordinary program with little extra effort.

2 Related Works

OpenCL provides rich low-level APIs for heterogeneous programming. It builds a solid foundation for high-level extensions. OpenACC [5] is a high-level directive based approach currently targets at single device. It reduces the difficulty of programming at the expense of performance and flexibility.

JSeriesCL [6] defines ParameterGPU class to simply the OpenCL execution and using associated data and thread attributes to decide the size of work-groups and work-items. Using it requires heavy prior training and the code has poor readability.

The SOCL [7] framework provides a unified OpenCL platform for multi-device system. It ease the restrictions on platforms, contexts and command queues of OpenCL without changing the development process.

The Skeleton computing language (SkelCL) [8] is a high-level extension of OpenCL for multi-GPU system. It introduces parallel container data types and parallel skeletons to realize automatic data distribution and parallel computation. Although it's powerful, it has a limited scope of applications. It's learning cost could be high.

The VirtCL [9] framework provides a single high-level abstraction for multiple devices. It implements a front-end library instead of the OpenCL APIs. Since the VirtCL is presented to solve the problems of memory inconsistency and device contention, its library only partly conceals the OpenCL programming details.

3 OCLS Framework

3.1 OCLS Abstraction Layer

The initial motivation of OCLS is to reduce the difficulties to exploit the massively parallel computing capability of heterogeneous system. Application developers demand a high-level abstraction as they are suffering from low-level programming. OCLS provides a unified abstraction layer between programmers and heterogeneous systems and implements such abstraction on the top of OpenCL as illustrated in Fig. 1. For programmers, they are interacting with a virtual processor through OCLS library. There is little difference from programming on a multi-core processor except that the parallel scale can be several orders of magnitude higher. They will concentrate on designing and implementing parallel algorithms into kernels and call the OCLS APIs to execute them. The execution is controlled by the parameters passed to the API function. All the details of low-level programming on various heterogeneous processors are handled implicitly. With C compatible library and OpenCL kernel language, OCLS integrates heterogeneous programming into ordinary programs.

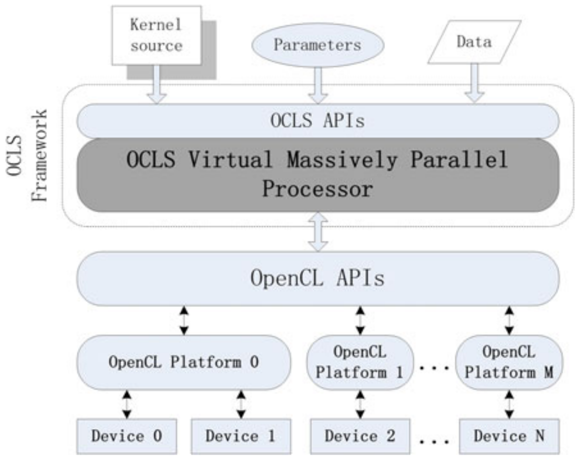

Fig. 1 OCLS abstraction layer

3.2 OCLS Library

The OCLS library consists of four primary functions and three assistant functions. It encapsulates OpenCL APIs and minimizes the development process of OpenCL. The four primary functions indicates the procedures in OCLS programming: initialization, kernel execution, execution finalization and termination.

The *ocl_Init()* function queries all available platforms and devices and selects a best device to initialize the environment configuration. It then compile the kernel source for execution. If the kernel source is stored in kernel files, call the assistant function *prog_Src()* to read them in ahead of *ocl_Init()*. It takes the kernel compile option as parameter and just need to be called once.

Programmers use the *ocl_Runkernel()* function to launch a kernel. They just give the name of the kernel and specify the execution scale (NDRange in OpenCL) along with the kernel parameters. *ocl_Runkernel()* then creates the kernel, set the kernel arguments and run it on the selected device.

The *ocl_Finkernel()* function finalizes the kernel execution. It sets a synchronization point and automatically handles the data movement. It should be called before the subsequent calculations using the results and need not to be paired with *ocl_Runkernel()*.

The *ocl_End()* function terminates the OCLS programming by releasing allocated memory space on both the host and the device. The shared OpenCL objects like context, program and kernel created earlier are also purged.

3.3 Kernel Data Type and Data Movement

OCLS creates a new kernel data type *ocl_kdata*. It consists of a pointer to the original data, the device side memory object, a data size variable and the I/O type variable. The I/O type includes five predefined value: NORMAL, TEMP, IN, OUT and INOUT which indicates the relationship between the kernel execution and data.

All the parameters of the kernel should be created in *ocl_kdata* type and initialized using assistant function wrapper(). The data pointer, data size and I/O type are specified by programmer when calling wrapper(), the device memory object is managed implicitly according to the I/O type. The NORMAL type states that the parameter is a regular variable. Others indicate device side memory allocation. IN and INOUT types state the parameter as the input data to the kernel, a implicitly data transfer from host to device is incurred. Device to host data movements are handled by *ocl_Finkernel()* after kernel execution on the data with OUT and INOUT type.

Kernel execution often needs extra memory space for intermediate results. Those parameters are stated as TEMP type. OCLS also provide assistant function *flush_Data()* for user controlled random data transfer for sake of flexibility.

3.4 Runtime Data Structures

OCLS introduces several data structures at runtime to provide convenience and eliminate redundant operations.

The *parallel capacity* refers to the product of the compute units amount and clock frequency of a device. It's the criterion that OCLS uses to select the best device.

After the kernel execution, kernel data with OUT or INOUT type is automatically pushed into a shared *output stack*. A ocl_Finkernel() function called later empties the stack and copies the results back.

The same kernel will be created in every call to function *ocl_Runkernel()* If it is launched repetitively. It also affects the output stack. A *history pointer* is introduced to solve this problem by recording the last executed kernel. If the same kernel is launched, *ocl_Runkernel()* will skip the kernel creation and pushing operations.

A *buffer list* is used to indicate the location of each buffer in device memory. When a buffer is created in *wrapper()*, a pointer to the buffer is added to the tail of the buffer list. When calling *ocl_End()*, it frees allocated memory space according to the list.

4 Case Study

An example of implementing the same vector addition algorithm in both OCLS and OpenCL is presented to demonstrate the use of OCLS and its advantages. Algorithm 1 show the kernel source code of vector addition which is used for both Algorithm 2 and 3. The initialization of the three integer vectors A, B and C with a same size specified by a variable *datasize* is omitted.

Algorithm 2 and 3 both archive the function of executing vector addition on a device in parallel. But the original OpenCL program has to specify the device type explicitly which limits its portability and needs an extra 29 lines in source code to accomplish the same task. It omits all the error handling which is done implicitly in OCLS and the readfile() function used to read in kernel file is also undefined. The example shows that using OCLS can reduce the programming effort significantly with user-friendly APIs and gives a brief look at the portability and reliability of OCLS.

Algorithm 1: Vecadd kernel

```
__kernel void vecadd(global int *a,\
global int *b,global int *c)
{
    int id = get_global_id(0);
    c[id] = a[id] + b[id] ;
}
```

Algorithm 2: Vecadd in OCLS

```
ocl_kdata da,db,dc;
prog_Src("kernel.cl");
ocl_Init(NULL);
da = wrapper(A,datasize,IN);
db = wrapper(B,datasize,IN);
dc = wrapper(C,datasize,OUT);
ocl_Runkernel("vecadd",1,&elements,NULL,3,da,db,dc);
ocl_Finkernel();
ocl_End();
```

Algorithm 3: Vecadd in OpenCL

```
cl_int err;
cl_platform_id platform;
cl_device_id device;
cl_device_type devicetype;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;
char * clProgStr = "";
cl_mem dA,dB,dC;
devicetype = CL_DEVICE_TYPE_GPU;
clGetPlatformIDs(1,&platform,NULL);
clGetDeviceIDs(platform,devicetype,1,&device,NULL);
cl_context_properties prop[] = {CL_CONTEXT_\
    PLATFORM, (cl_context_properties) platform, 0};
context = clCreateContext(prop,1,&device,NULL,NULL,\
    &err);
queue = clCreateCommandQueue(context,device,0,&err);
clProgStr = {readFile("kernel.cl")};
program = clCreateProgramWithSource(context,\
    1,(const char **) &clProgStr, &srcLength, &err);
clBuildProgram(program,1,&device,NULL,NULL,NULL);
dA = clCreateBuffer(clContext,CL_MEM_READ_ONLY\
    ,datasize,NULL,&clStatus);
dB = clCreateBuffer(clContext,CL_MEM_READ_ONLY\
    ,datasize,NULL,&clStatus);
dC = clCreateBuffer(clContext,CL_MEM_WRITE_\
    ONLY,datasize,NULL,&clStatus);
clEnqueueWriteBuffer(queue,dA,CL_TRUE,0,datasize,\
    &A,0,NULL,NULL);
clEnqueueWriteBuffer(queue,dB,CL_TRUE,0,datasize,\
    &B,0,NULL,NULL);
clEnqueueWriteBuffer(queue,dC,CL_TRUE,0,datasize,\
    &A,0,NULL,NULL);
clSetKernelArg(kernel,0,sizeof(cl_mem),(void*)&dA);
clSetKernelArg(kernel,0,sizeof(cl_mem),(void*)&dB);
clSetKernelArg(kernel,0,sizeof(cl_mem),(void*)&dC);
clEnqueueNDRangeKernel(queue,kernel,1,NULL,\
    &elements,NULL,0,NULL,NULL);
clEnqueueReadBuffer(queue,dC,CL_TRUE,0,datasize,\
    &C,0,NULL,NULL);
Free(clProgStr);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(dA);
clReleaseMemObject(dB);
clReleaseMemObject(dC);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

5 Evaluation

All of the evaluations were conducted on both a GPU server with two Xeon E5520 CPUs, 12G RAM and four NVIDIA Tesla C1060 GPUs running CUDA 6.5 [10] and CentOS 6.5 with linux Linux 2.6.32-431 kernel and a Intel MIC server with two Xeon E5-2670 CPUs, 64G RAM and two Xeon Phi 7110P co-processors running Intel OpenCL runtime 14.2 with MPSS 3.3.4 and Red Hat Enterprise Linux Server 6.3 with linux 2.6.32-279 kernel. The OpenCL version on both platform is OpenCL 1.2. The benchmarks we used were collected from the Parboil benchmark suite [11]. The benchmarks and corresponding data sets are shown in Table 1.

Table 1 Benchmarks and data sets

Benchmark	Problem size (small)	Problem size (large)
bfs	270,926 nodes	1,000,000 nodes
cutcp	5943 atoms	–
histo	996 w × 1040 h 20 iterations	–
lbm	2.16×10^6 cells 100 iterations	–
mri-q	32768 pixels using 3072 samples	262144 pixels using 2048 samples
sgemm	Matrix size: 128 × 96 96 × 160	Matrix size: 1024 × 992 992 × 1056
stencil	Grid size: 128 × 128 × 32	Grid size: 512 × 512 × 64

5.1 Code Size Comparison

All the benchmarks were rewrote using OCLS without any changes or optimization to the algorithm. Figure 2a shows the source code size of the benchmark and OCLS programs. Figure 2b shows the normalized source code size comparison. The reduction in coda size is related to the program feature since OCLS only reduces the code of parallel relevant parts. Benchmark *cutcp* spends much effort on data processing and serial computing thus produces the lowest reduction ratio. OCLS archives an average 8.97 KB reduction in code size. The average reduction ratio is 35.4 %.

5.2 Performance

The OpenCL programs in Parboil benchmark suite target GPU. It needs some manual modification in environment configuration to get them run on MIC platform. However, benchmark *cutcp* kept failing on GPU due to the OpenCL implementation issues in CUDA, it was removed during the GPU test. Benchmark *histo* incurred hardware exception and segmentation fault during the MIC test, the program returned with incorrect result. However, the OCLS version of *histo* ran properly.

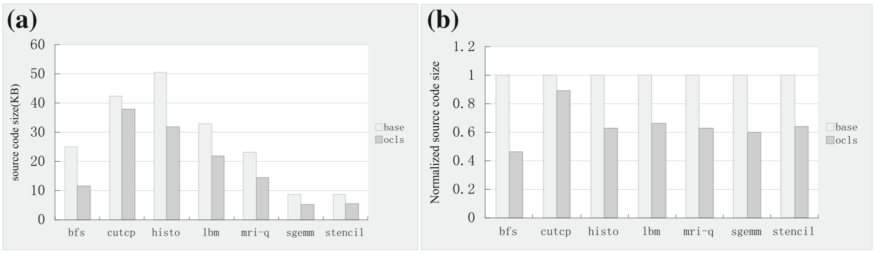

Fig. 2 Source code size comparison (*base* original OpenCL program)

Although OCLS introduces overhead in device querying and extra data structures creation, it eliminates redundant operations introduced by encapsulation with the runtime data structures and avoids unnecessary data transfer in original benchmark program. It yields better performance on both GPU and MIC as shown in Fig. 3. Benchmark *cutcp*, *histo*, *lbm*, *mri-q* used *small* data set. Benchmark *bfs*, *sgemm*, *stencil* used *large* data set.

5.3 Stability

We also evaluated the stability of OCLS programs using different size data sets provided by the benchmark suite. Figure 4 illustrates the test result. We can see that the performance of OCLS is always better than original OpenCL programs. The trend in the execution time changes of OCLS follow that of OpenCL. OCLS performed as stable as OpenCl with different problem size and kept its advantages. It's also very interesting to find the differences between GPU and MIC of their behavior with different data size and application.

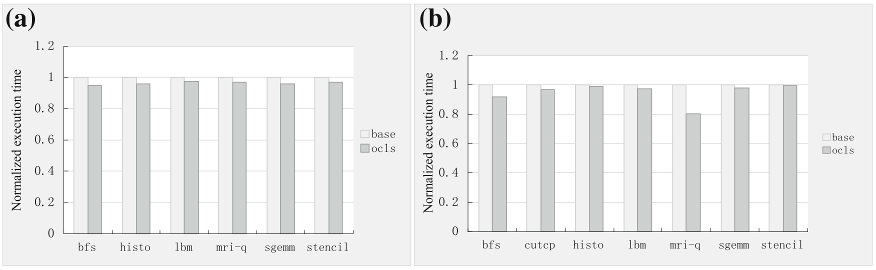

Fig. 3 Normalized execution time comparison

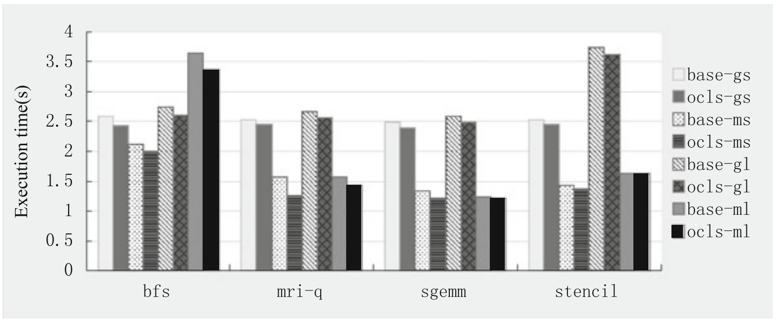

Fig. 4 Execution time with different data sets (-s: small, -l: large) on GPU and MIC (-g: GPU, -m: MIC)

6 Conclusion

We have proposed a simplified high-level programming framework called OCLS for heterogeneous system. It allows programmer to exploit the massively parallel computing capability of various heterogeneous processors in an ordinary program without concerning the hardware details. This is archived through the OCLS library which encapsulates the OpenCL APIs. The OCLS library minimizes the development process and realizes automatic environment configuration and data movement. The comparison with original OpenCL programs shows that using OCLS can reduce the amount of code and the programming effort significantly. In the experimental evaluation on GPU and MIC with different data size, OCLS showed a stable and better performance than the benchmarks. The experiments also demonstrated the portability and stability of OCLS.

Acknowledgments This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No.61173039, and the National High Technology Research and Development Program (863 Program) of China under Grant No. 2012AA010904.

References

1. Top500.org. http://www.top500.org/
2. Javier Diaz, Camelia Munoz-Caro, Alfonso N (2012) A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans Parallel Distrib Syst 23(8):1369–1386
3. Brodtkorb Andre R, Christopher Dyken, Hagen Trond R et al (2010) State-of-the-art in heterogeneous computing. Sci Program 18:1–33
4. The OpenCL specification. https://www.khronos.org/opencl/
5. OpenACC–directives for accelerators. http://www.openacc-standard.org/
6. de Souza Rosa Gomes R, Figueiredo JM, Martins CA et al (2014) A framework for automating the configuration of OpenCL. Environ Model Softw 53:81–86
7. Henry S, Denis A, Barthou D, Counilh M-C, Namyst R (2014) Toward OpenCL automatic multi-device support. In: Euro-Par 2014, LNCS, vol 8632. Springer, Heidelberg, pp 776–787
8. Steuwer M, Gorlatch S (2014) SkelCL: a high-level extension of OpenCL for multi-GPU systems. J Supercomput 69:25–33
9. You Y-P, Wu H-J, Tsai Y-N et al (2015) VirtCL: a framework for OpenCL device abstraction and management. In: 20th ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 161–172
10. CUDA toolkit. https://developer.nvidia.com/cuda-toolkit
11. Parboil Benchmarks. http://impact.crhc.illinois.edu/Parboil/parboil.aspx