Mapping Streaming Applications to OpenCL

Abhishek Ray[*1]
[*]Student Author [1]Nanyang Technological University

ABSTRACT
Graphic processing units (GPUs) have been gaining popularity in general purpose and high performance computing. A GPU is made up of a number of streaming multiprocessors (SM), each of which consists of many processing cores. A large number of general-purpose applications have been mapped onto GPUs efficiently. Stream processing applications, however, exhibit properties such as unfavorable data movement patterns and low computation-to-communication ratio that might lead to a poor performance on a GPU. We describe the automated mapping framework developed earlier that maps most stream processing applications onto NVIDIA GPUs efficiently by taking into account its architectural characteristics. We then discuss the implementation details of porting the mapping framework to OpenCL running on AMD GPUs and evaluate the performance of the mapping framework by running several benchmarks. Performance between the generated CUDA and OpenCL code is compared based on different heuristics.

1. INTRODUCTION

A GPU consists of a number of multi-threaded processors called streaming multiprocessors (SM). Each SM consists of execution cores, which execute in SIMD mode. On each SM, threads are grouped into scheduling units called wavefronts and warps, respectively, in AMD and NVIDIA terminology. Several warps/wavefronts can be supported on each SM at the same time.

Streaming applications are an important domain of applications. Streaming applications can be represented as graphs, which are composed of nodes that communicate independently over data channels [1][6].

An automated mapping framework [2] was developed that maps stream processing applications onto GPUs efficiently by taking into account architectural characteristics of NVIDIA GPUs. The mapping flow captures the parallelism from the streaming language, models the parallel architecture of the GPU and uses a novel execution model of heterogeneous threads, which is basically a mix of compute and memory access threads.

1.1 Motivation

CUDA and OpenCL are two different programming frameworks for programming GPUs. CUDA is NVIDIA's proprietary technology and is specific to NVIDIA GPUs whereas OpenCL is an open and free standard managed by the Khronos Group [4] that can be used to program different devices such as CPUs, GPUs and DSPs. Its portability is one of the prime motivations for porting framework to OpenCL.

2. DESIGN

The mapping framework is divided into two parts:

StreamIT language – In StreamIt, the basic programmable unit is called a Filter. It has a single input and a single output and its body is essentially Java-like code. StreamIt programs have a hierarchical graph structure where a filter is represented by a leaf node. The flow of StreamIt programs can be further distributed by placing filters into any of the composite blocks such as pipelines, splitters and joiners.

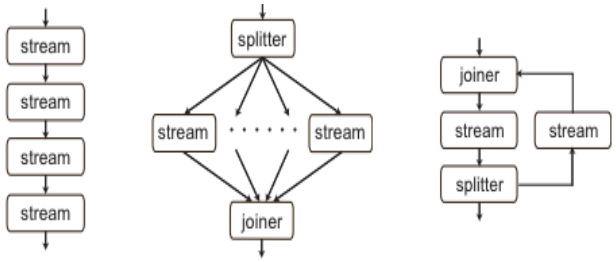

Figure 1: Different Hierarchical Streams provided by StreamIT

Automated Mapping onto GPUs –

A filter executes in a sequence of steps. Firstly, it reads the data from the memory. Secondly, it performs the computation on the data and lastly, it writes it back to the memory to pass it to the next processing filter. Usually the ratio between computation and memory access is small. Therefore, if global memory is used to store the filter's input and output data, most of the time will be spent on memory accesses. Thus, it is beneficial to bring this data onto the shared memory as the threads can access this region of memory faster.

In the framework two different types of threads are used: memory access (M) threads and compute (C)

threads. M threads prefetch the data for the next stream execution while the C threads perform computation on the data fetched by M threads onto the shared memory in the previous execution.

As C threads are always ready for execution, they can access the SM.

The automated mapping framework transforms the code in following ways:

- Memory transfer operations with large latency are clustered into dedicated threads.
- The data flow is transformed based on the parallelism exhibited by StreamIt.

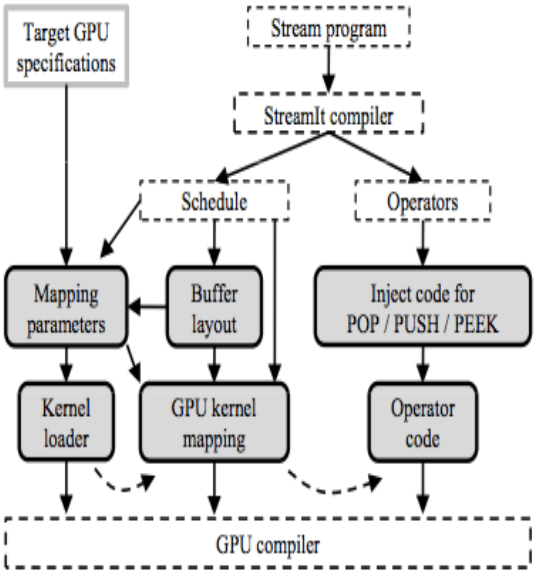

Figure 2: Automated Mapping Flow of the Framework

The mapping framework is implemented as an extension to the back-end of the StreamIt compiler. C code is generated by the application that can be compiled by the GPU compiler. The StreamIT compiler flattens the hierarchical stream graph and generates a schedule, which consists of the order and the number of executions of each of these operators. After StreamIT generates the schedule, the mapping framework takes over as the generated schedule is sequential and is meant for single-threaded execution.

The requirements of each operator in the schedule are analyzed and a buffer layout is produced. After this various mapping parameters such as the number of stream schedules to execute in parallel, the number of C threads required for the execution of each stream schedule, and the number of M threads accessing global memory are determined according to the stream schedule structure and specification of the target GPU.

Two components are built after this – A Kernel loader, which runs on the CPU and performs all the initializations and the GPU kernel code that executes on the device and performs the mapping and computation.

3. MAPPING To OpenCL

Porting an application from CUDA to OpenCL is straightforward as both programming frameworks have similar syntax [3]. However, there are a few major differences.

In CUDA, both the host code and the device code is compiled at the run time where as in OpenCL host code is compiled statically and the device code is compiled at run time. As a result, an additional header file is automatically generated in OpenCL, which stores all the macros, which are required, by both the host code and the device code. These macros were defined in the device code in CUDA.

Another key difference is in the device initialization. Since OpenCL targets a lot of different platforms, it has a complicated initialization process as compared to CUDA.

Table 1: Syntax Difference – Device Initialization

CUDA	OpenCL

Schedule<<<grid, threads, sharedSize>>>(in_stream, out_stream, thread_step, iterations)	status = clGetContextInfo(context, CL_CONTEXT_DEVICES, ..);
	commandQueue=
	
	clCreateCommandQueue(context, devices[0], ..);
	
	in_stream = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, ..);
	
	out_stream = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,..);
	
	clBuildProgram(program, 1, devices, NULL, NULL, NULL);
	
	kernel = clCreateKernel(program,

	"Schedule", &status);
	
	clSetKernelArg(kernel,0, sizeof(cl_mem), (void *)&in_stream);
	clEnqueueNDRangeKernel(commandQueue, kernel, ..);

Table 2: Syntax Difference- Kernel Code

CUDA	OpenCL

Global void Schedule (unsigned int *in_stream, unsigned int *out_stream) {
 	
extern __shared _____unsigned int volatile shared_mem[];
 	
int iterations_start = iterations_total * **blockIdx.x** / **gridDim.x**; int iterations_stop = iterations_total * (blockIdx.x + 1) / gridDim.x; if ((threadIdx.x < WORKERS) && (**threadIdx.x** % 32 == 0)) sync_work[threadIdx.x / 32] = 0;
 	
_____**syncthreads();**
 	
}
	kernel void Schedule(_____global unsigned int * in_stream, ____global unsigned int * out_stream, const int thread_step,const int iterations_total, **__local volatile unsigned int shared_mem**)
	
	{
	
	int iterations_start = iterations_total ***get_group_id(0)** / **get_num_groups(0)**; int iterations_stop = iterations_total * (get_group_id(0) + 1) / get_num_groups(0); if ((get_local_id(0) < WORKERS) && (**get_local_id(0)** % 32 == 0)) sync_work[get_local_id(0) / 32] = 0;
	
	barrier(CLK_LOCAL_MEM_FENCE);
	
	}

4. EXPERIMENTS

After mapping the streaming application to OpenCL, different benchmarks were tested to check for correctness of the generated code and to compare the results of the OpenCL code with the CUDA code. The devices compared were AMD Radeon HD 6970 running OpenCL and NVIDIA Tesla C2050 running CUDA.

Table 3: Architectural Differences between devices being compared

	AMD Radeon HD 6970	NVIDIA Tesla C2050
#Compute Unit	24	14
#Cores	-	448
#Processing Elements	1536	-
#Core Clock (MHz)	800	1500
#Max BW (GB/s)	176	144
#Max GFLOPs	2703	1288

Parameters used for the Graphs:

- S, the number of C threads per execution.
- F, the number of M threads that transfer data between global and SM memory.
- X – axis – number of parallel stream executions in each SM.
- Performance is measured in FLOPS [5]. It is calculated as the number of floating point operations / second.
- Performance is compared between OpenCL and CUDA by taking into account their respective FLOPS using the same parameters. It is calculated by $FLOPS_{OPENCL}$ / $FLOPS_{CUDA}$ * 100

4.1 PERFORMANCE COMPARISON BETWEEN GPU & CPU

Performance between the GPU (Radeon HD 6970) and the CPU (Intel Xeon CPU E5540 @ 2.53 GHz) was compared. As expected, the GPU easily outperformed the CPU.

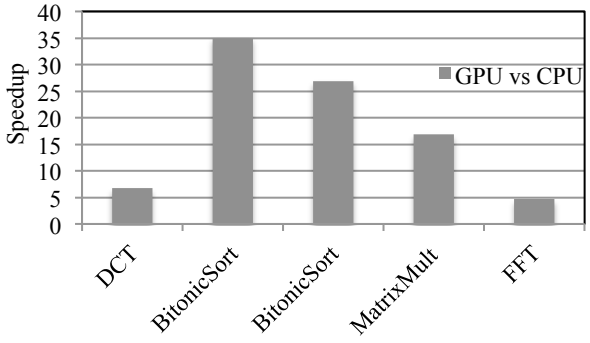

Figure 3: Performance Comparison – GPU vs CPU

4.2 PERFORMANCE COMPARISON BETWEEN OPENCL & CUDA

Performance between OpenCL and CUDA was compared by taking into account the FLOPS (Floating

Point Operations/Second).

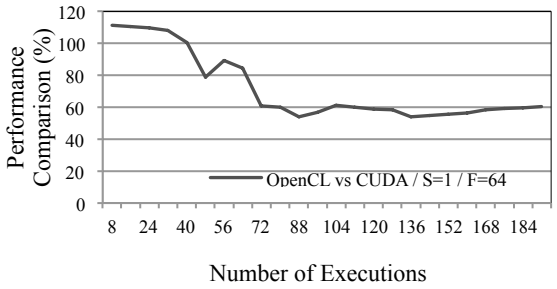

Figure 4: Performance Comparison – Bitonic Recursive – Radeon HD 6970 & Tesla C2050

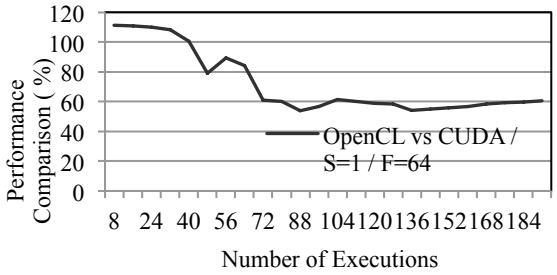

Figure 5: Performance Comparison – DCT – Radeon HD 6970 & Tesla C2050

OpenCL initially performs better than CUDA till it experiences a drop in performance. This drop in performance is because at this point CUDA experiences a full warp (32 threads) and hence an increase in performance. At this point, the performance of OpenCL in comparison to CUDA drops. After this point, the performance of OpenCL increases linearly in comparison to CUDA as OpenCL attains a full wavefront (64 threads). However, performance of CUDA on Tesla C2050 increases at a faster rate in comparison to Radeon HD 6970 as it has a higher core clock.

4.3 FAIR PERFORMANCE COMPARISON BETWEEN OPENCL & CUDA

As we are comparing performance across two different GPU architectures, it isn't fair to just compare the FLOPS, which takes into consideration the execution time of the application that depends on how powerful the processor it. If the comparison had been made on the same device, then comparing the performance based on FLOPS would have been fair.

Performance between OpenCL on Radeon HD 6970 and CUDA on Tesla C2050 is compared fairly by taking into account factors, which are most important for our application. These factors are listed Memory Bandwidth, Floating Point Performance and Architecture Related Differences.

Table 4: Comparison between the two devices based on different heurestics for FFT benchmark

	AMD Radeon HD 6970	NVIDIA Tesla C2050
#BW attained (GB/s)	0.0105	0.0288
#Flops attained (GFLOPs)	0.1124	0.3079

As we can see from Tables 3 and 4, even though Radeon HD 6970 has a higher number of cores, a higher theoretical floating-point performance and a higher theoretical memory bandwidth in comparison to Tesla C2050, it achieves a lower performance. We see that both devices attain very low GFLOPs. The main reason for this is that both devices attain a very low memory bandwidth and hence this limits the performance, as floating-point performance is directly proportional to the memory bandwidth. An application cannot experience an increase in floating point performance if it is blocked waiting for data due to insufficient memory bandwidth.

Performance of the mapping framework on CUDA was evaluated after removing the data from the constant memory and putting it into global memory. As expected, there was a decrease in performance as compared to the benchmarks, which used constant memory.

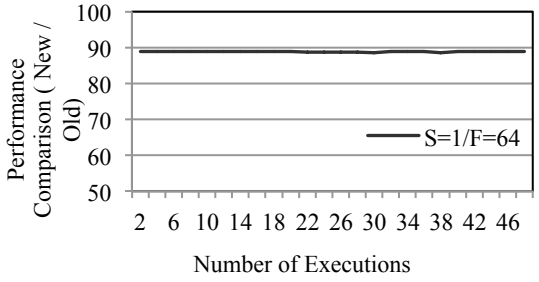

Figure 6: Effect Of Constant Memory on Performance – DCT

Performance of the application drops after the removal of Constant Memory in CUDA as expected but the performance drop is just about 10-12%, which isn't a very significant drop, and its performance is still better than OpenCL.

Performance of the mapping framework was then evaluated by estimating the effect of S on the different devices. Performance of OpenCL doesn't increase at the same rate as CUDA as S is increased.

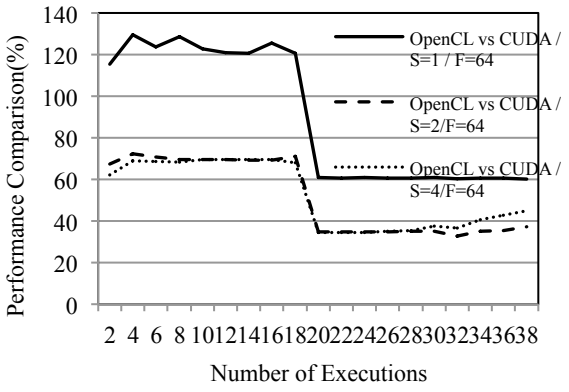

Figure 7: Effect of S on the Performance – MatrixMult

The performance of OpenCL on Radeon HD 6970 increases slower than CUDA on Tesla C2050 when S is increased. This shows that as the degree of parallelism is increased, CUDA responds much better than OpenCL. CUDA exposes the parallelism better than OpenCL.

One of the main reasons why the performance of OpenCL decreases slower than CUDA when S is increased is because Radeon HD 6970 has a slower core clock as compared to Tesla C1060 and Tesla C2050 as seen in Table 5. Core clock gives an indication of how fast the processor operates on each GPU and hence when we increase S, the device has to process more data and Tesla C2050 processes this data faster than Radeon HD 6970.

OpenCL on AMD is a portable language for GPU programming as it can target different devices. However, there might be a performance penalty because of its portability. CUDA seems to be a better choice for applications where achieving as high a performance as possible is important.

5. CONCLUSION

Different benchmarks were tested and their performance on OpenCL analyzed after the framework was successfully ported.

Performance between OpenCL and CUDA GPUs was initially evaluated and analyzed by comparing the FLOPS attained. However, as FLOPS is a measure of execution time, it doesn't ensure fairness. A new fair method of comparison was developed which takes into consideration the architecture of the different devices being compared. The comparison was made using a few different heuristics most important for the mapping framework.

6. ACKNOWLEDGMENTS

I would like to thank Dr. Huynh Phung Huynh and the staff at the Institute Of Higher Performance Computing and Prof. Stephen Turner and the staff at PDCC Lab in NTU for their continued support throughout the year.

7. REFERENCES

[1] M. Gordon, *"Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures."*

[2] Hagiescu, A.; Huynh Phung Huynh; Weng-Fai Wong; Goh, R.S.M.; , *"Automated Architecture-*Aware Mapping of Streaming Applications Onto GPUs," *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* , vol., no., pp.467-478, 16-20 May 2011

[3] http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx

[4] The OpenCL Specification. - http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[5] http://en.wikipedia.org/wiki/FLOPS

[6] W. Thies, *"Language and Compiler Support for Stream Programs."*

[7] Huynh Phung Huynh, Andrei Hagiescu, Weng Fai Wong, Rick Siow Mong Goh. *Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* Feb 2012