Accelerating High-Throughput Computing through OpenCL

Andrei Dafinoiu, Joshua Higgins, Violeta Holmes
High-Performance Computing Research Group
University of Huddersfield
Huddersfield, United Kingdom

***Abstract*—As the computational trend diverges from standard CPU computing, to encompass GPUs and other accelerators, the need to integrate these unused resources within existing systems becomes apparent. This paper presents the implementation of a HTCondor pool with GPU execution capabilities through OpenCL. Implementation is discussed from both the system setup and the software design standpoint. The GPU landscape is investigated and the efficiency of the system is evaluated via Fast-Fourier Transform computations. Experimental results show that HTCondor GPU performance matches a dedicated GPU cluster.**

I. Introduction

In more recent years, the computing environment has seen a shift from the traditional CPU based computing onto a much divese, and more parallel architecture such as GPUs. As of November 2015, two of the top ten supercomputers make use of GPU accelerated computing[1].

However, not all computational tasks are based on raw execution performance. Thus the emergence of High-Throughput Computing (HTC), a branch focusing on computational tasks that require the use of resources over an extended period of time.

The HTC community is not concerned about the execution rate of jobs, but rather, the parallelism of discrete jobs, given a much larger pool of resources. The interest is in how many jobs can be executed over a given amount of time, not the execution speed of a single job. HTC can take advantage of opportunistic resources, for example idle PCs in a University campus, to execute tasks, thus ”stealing” CPU time[2].

OpenCL is a heterogeneous programming framework for the development of applications that span across multiple architectures, including CPUs, GPUs, DSPs, and other accelerators. Whilst, from a computing standpoint, GPUs are being used as accelerators in supercomputers, the commodity, general purpose GPUs (GPGPUs) available in typical workstations may not have a significant impact on the fast processing required for High Performance Computing. In HTC, the fast performance of individual units is not the most important consideration, thus making HTC a better candidate for GPGPU based acceleration[3].

Previous work has shown that middleware vendors support the idea of GPU computing in HTC through built-in detection methods of GPU capabilities, as it is the case with newer versions of HTCondor[4]. Typically this relies on CUDA, the proprietary GPU computing framework developed by NVidia. However, CUDA is limited to newer NVidia GPUs, while also requiring special packages to be installed on the system in order to function[5]. In terms of general purpose PCs available on a university campus, only a small number of workstations may be capable of supporting CUDA. In order to exploit a much larger and more diverse mix of computing resources, OpenCL is a framework that can be used to exploit a much larger pool of devices than CUDA.

While attempts have been made to create computing environments using commodity GPUs, these implementations typically operate as dedicated clusters with the sole purpose of emulating typical GPU clusters, thus requiring large investments to develop[6].

A significant number of UK universities deploy HTC pools through HTCondor, including Oxford Unversity, Cambridge University, and Manchester University[7], however there is limited research output indicating the development of GPU integration within these pools.

This research aims to expand the capabilities of the existing HTC system by enabling use of the GPU resources in addition to the CPU resources of the pool, to be utilised during periods when they would otherwise be idle, supplementing the dedicated GPU computing cluster. To evaluate the ease-of-use, efficiency and flexibility of the OpenCL framework across the heterogeneous HTCondor pool, an FFT computation test case is implemented.

By increasing the amount of GPU resources available to researchers it is anticipated that the system would be more appealing, and GPU computing would be more accessible. This, in turn, would encourage some of the researchers using CPU intensive programs to shift to GPU instead, and improve utilisation of dormant resources.

II. Background

A. HTCondor

HTCondor is a workload management system used primarily for executing processes in a opportunistic environment, where tasks are distributed between resources as they become available, allowing the user to take advantage of otherwise idle resources. It features execution queues, job scheduling, priority, and resource discovery and management[2].

HTCondor employs cycle-stealing, the ability to use general workstations whilst they are unused for their specific purpose and would otherwise sit idle. HTCondor also contains a checkpoint system that allows it to migrate jobs to different machines once a machine starts to be used by the owner.

Submitted jobs are matched to resources by using the ClassAd mechanism, a framework that allows both jobs and machines to specify requirements and or preferences in regards to resource allocation.

The University of Huddersfield implements a HTCondor pool across all workstations available on campus, ranging from low-end library dedicated PCs to high-performance workstations in the design laboratories[7]. The university has a policy in place setting the minimum core count of a processor to 4, however no restrictions apply to the GPU components. The system is accessed via SSH authentication on the HTCondor headnode, and user data storage is offered via a GLUSTER based mirrored storage.

The HTCondor pool actively updates available resources, with offline machines being removed from the HTCondor pool. There are approximately 7000 CPU cores registered within the pool, however, the number of available cores changes constantly based on the availability of PCs on campus. On average, between 700 and 3000 are available for opportunistic jobs at peak times.

B. OpenCL

OpenCL is an open source framework for developing heterogeneous programs, created by a consortium of leading companies to develop standards for graphic acceleration and parallel computation, which has grown to include most commodity CPU/GPU providers, as well as specialised accelerators in the form of FPGAs and DSPs[3].

Since OpenCL has wide device support, workstation specifications are not as relevant at the programming stage of development, making OpenCL a viable solution for environments that contain more than one architecture or operating system. Its ability to operate using base drivers promises a much faster integration with existing systems.

Development of OpenCL applications is split into two different files. The host file is a C code that deals with the outer control logic of the system, dispatching work kernels to the compute units, controlling memory read/writes and executing the serial segments of the code. The kernel is also written in C, however it also incorporates OpenCL specific syntax. It represents the parallel component of the program, to be executed on the compute nodes, and can be compiled at run time by the host, to allow for a more diverse execution environment.

The flexibility of OpenCL allows for the creation of programs targeting machines with CPUs and GPUs (used individually or together), CPUs and other accelerators, or dedicated clusters that contain multiple GPUs (or CPUs) within each node.

III. OpenCL on HTCondor

In order to evaluate feasibility of OpenCL within a HTCondor pool, a test case was designed around the use of Fast-Fourier Transforms, as they are the basis for many computational algorithms in scientific software[8].

A. Configuring Condor

A HTCondor job must be scheduled for execution within a 'slot'. The default implementation advertises separate slots for each CPU core in a machine. The HTCondor ClassAd could be modified to add information about the GPU resource. HTCondor recommends that the GPU is appended to one of the CPU slots,[9] so that a machine may run both OpenCL and regular CPU jobs at the same time, as can be seen in Figure 1. During preliminary testing of this environment, it was determined that such an implementation slows down the OpenCL execution, for example where multi core CPU parallelism is employed, hence a different approach was considered.

Therefore, we configured a slot that allows an OpenCL program to block the entire machine, using all available resources. This was done alongside the existing slots, allowing a computer to either act as a CPU resource, advertising each CPU node individually, or as a single OpenCL entity, as shown in Figure 1. This dual setup is mutually exclusive, meaning that a machine that has started running a CPU job will be unavailable for OpenCL jobs, and vice versa.

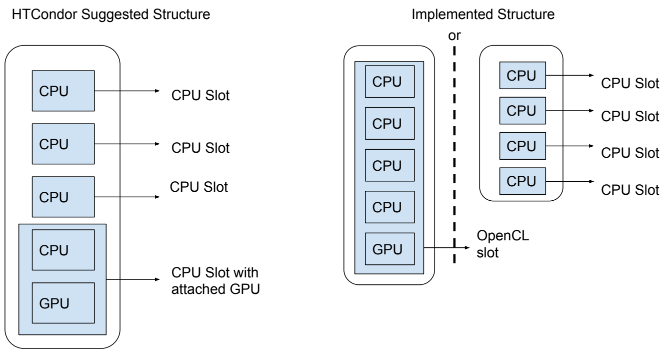


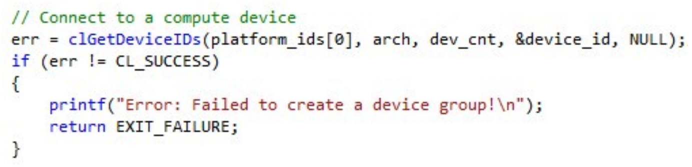
Fig. 1. HTCondor Slot Structure

B. Implementing OpenCL

The host file is created in order to setup the OpenCL environment, control execution and manage data transfers. Multiple segments of the host file are reusable across different OpenCL programs, mainly those that deal in environment setup and data transfers. Memory sizes, work-flow and optimisations depend on the targeted architecture and device. Within the host file, the command queue which controls execution is defined. It creates the execution kernels and enqueues them, either sequentially or parallel, also transferring memory buffers between devices and the host.

The kernel file, executed on the compute device, represents the parallel segment of the program. It encapsulates the computations to be executed and is directly controlled by the host. The kernel file does not change across different architectures or devices, and is only influenced by the type of parallelism used. The kernel file functions as a C function, that is executed inside a loop, either synchronously or asynchronously. Kernels can be task parallel, executing multiple functions simultaneously or data parallel, executing a single function on multiple data elements.

There are two methods of compiling the kernel file. The file can be compiled before execution, at the same time as the host file, reducing setup latency but limiting execution to the chosen architecture and device. Alternatively, the host file can be allowed to compile the kernel at run time to allow for execution across multiple devices and architectures. Figure 2 illustrates the compute device selection segment of the host code. The arguments passed to the function determine the selected device. As an example, assuming that a machine has both a CPU and a GPU, to execute the kernel on one or the other only requires a change in the arch variable in Figure 2. In reality, this results in a functional, yet unoptimised program. However, this example emphasises the flexibility of the OpenCL.



```
// Connect to a compute device
err = clGetDeviceIDs(platform_ids[0], arch, dev_cnt, &device_id, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to create a device group!\n");
    return EXIT_FAILURE;
}
```

Fig. 2. OpenCL host code excerpt

C. *GPU discovery and landscape*

HTCondor implements a function for GPU detection, however this relies primarily on CUDA and while also offering minimal support for OpenCL, returns insufficient device details to make optimisation decisions. For this reason, a program was designed to poll the target computer for available OpenCL devices and record their specifications.

Since OpenCL can run on CUDA drivers, there is no need to implement a separate CUDA program. The detection program was executed on 1000 machines, and, as seen in Table I, the GPU landscape discovered is quite diverse. Of the investigated machines, 300 failed to return information about their GPUs. This could be due to a lack of up-to-date drivers supporting OpenCL, or unsupported GPUs. The majority of devices are mid-range GPGPUs, mostly released around 2011. However, the program also identified a number of much newer generation NVIDIA GPUs, which feature a large number of cores that are more suited for efficient parallel execution.

D. *Application*

The Fast-Fourier Transform technique is used to convert time-domain signals into frequency-domain signals, and is widely adopted by researchers in engineering and science. For this reason it was chosen as a test-case for the system, which will be used in the aforementioned fields[8].

To better represent a real-world use case, GPU benchmarking was executed in a live environment, meaning that HTCondor resources became available only when idle, and jobs stopped when users returned to their machines. Tests were run using using the OpenCL Fast-Fourier Transform library on

TABLE I
GPU LANDSCAPE

GPU	**Nr**
AMD 5600	8
NVIDIA Quadro K600	42
NVIDIA GTX 610	40
NVIDIA GTX 670	75
NVIDIA GTX 750 Ti	77
NVIDIA GTX 970	133
AMD 6500	137
AMD 6400	189
Not detected	299
Total	**1000**

single dimensional FFTs[10]. To exploit the high number of available compute cores present in GPUs, FFT calculations were batched together until they filled the available memory size. The results are displayed in GFLOPs, calculated based on the FFTW benchmarking methodology[11]. To ensure the accuracy of the results, each FFT calculation was iterated 1000 times.

The challenge when using HTCondor is that the resource used for execution is unknown in advance, thus increasing the difficulty of optimizing the application for each individual GPU found through the discovery. For this experiment, the lowest GPU specification was used, to ensure execution across the entire system. For example, executing the program with a batch size lower than the GPU maximum will not exploit the best performance, however, using a batch size higher than the maximum will prevent execution.

Initial testing of CPU performance has shown between 0.6 and 1.7 GFLOPs on a standard Intel i5 CPU, shown in Figure 3. CPU benchmarking of the FFT implementation was not executed on a similar scale to the GPU since it would be beyond the scope of this work. More extensive CPU benchmarking has already been carried out by [12].

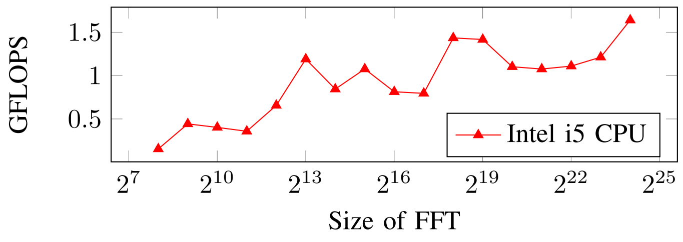


Fig. 3. CPU FFT execution

IV. Benchmarking Results

A. *Fast-Fourier Transforms on HTCondor GPUs*

The performance diversity of the GPU landscape can be noted in Figure 4, with the fastest GPU being an NVIDIA GTX 970 GPU, released in 2014, and the slowest being an AMD 5600 GPU, released in 2009. Newer generation GPUs benefit from higher clock speeds, more internal cores, and bigger memory buffers.

The average execution duration of the entire benchmark was 50 minutes. The first 40 % of jobs finished within 12 hours,

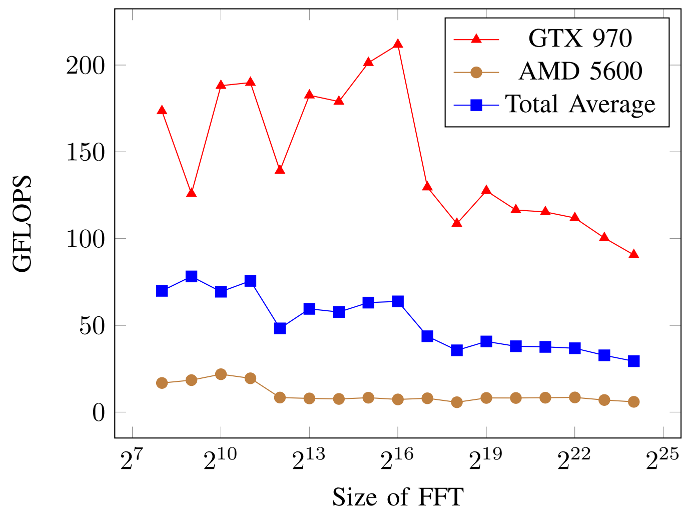


Fig. 4. GPU Benchmarking across 700 machines

with the other 60 % taking as long as 24 hours to complete. This is due to the fact that a job will be preempted when a real user accesses the machine.

B. Comparison with dedicated GPU cluster

In order to compare the available compute power of the GPUs inside HTCondor, the system needs to be compared against an existing GPU cluster. As such, the same benchmark program was executed on the NVIDIA C2050 GPU compute module inside VEGA, the dedicated GPU cluster at the UoH. In Figure 5, it can be seen that the average Condor GPU equals the performance of the single GPU compute module. However, the AMD GPUs are significantly slower overall, whilst some NVIDIA GPUs based on a newer architecture than the compute module achieve better performance.

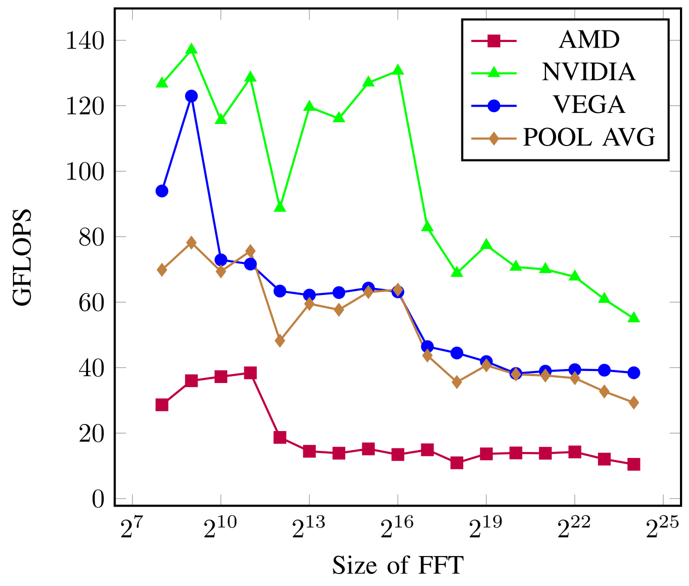


Fig. 5. Average Results by GPU Manufacturer

V. Summary

The successful integration of HTCondor with OpenCL has revealed a multitude of GPU resources that can be used to increase overall system performance for parallelizable applications, while also making GPUs more accessible in terms of physical usage and programming.

This work has shown that newer generation GPGPUs are able to match the performance of older dedicated GPU resources. However, due to the opportunistic nature of HTCondor, maximizing performance across such a system is difficult.

The flexibility and ease-of-use of OpenCL make it a promising framework for developing GPU applications across a diverse, and constantly evolving, environment.

VI. Future Work

Further develop the HTCondor implementation to facilitate access to GPU resources, by using the ClassAd system to advertise more GPU specific information for each machine, allowing users to optimise implementations for specific GPUs. Also, by exploiting the HTCondor ranking system, execution can be prioritised on machines with better capable GPUs foremost.

Improve performance of OpenCL applications by automating optimisations within the host file to increase performance within the HTCondor environment, whilst also documenting best-practise approaches and device-specific optimisations.

Acknowledgment

The authors would like to thank the QueensGate Grid at the University of Huddersfield

References

[1] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, no. 5, pp. 7–17, 2011.

[2] M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP journal*, vol. 11, no. 1, pp. 36–40, 1997.

[3] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[4] "condor gpu discovery." http://http://research.cs.wisc.edu/htcondor/manual/current/condor_gpu_discovery.html. Accessed: 2016-01-16.

[5] C. Nvidia, "Programming guide," 2008.

[6] S. Guba, M. Őry, and I. Szeberényi, "Harnessing wasted computing power for scientific computing," in *Large-Scale Scientific Computing*, pp. 491–498, Springer, 2013.

[7] D. Gubb, "Implementation of a condor pool at the university of huddersfielod that conforms to a green it policy," Master's thesis, University of Huddersfield, 7 2013.

[8] V. U. Reddy, "On fast fourier transform," *RESONANCE*, vol. 3, no. 10, pp. 79–88, 1998.

[9] "Htcondorwiki: How to manage gpus in series seven." https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToManageGpusInSeriesSeven. Accessed: 2015-12-03.

[10] "clfft: Opencl fast fourier transforms(ffts)." http://clmathlibraries.github.io/clFFT/. Accessed: 2016-02-11.

[11] "Fft benchmark methodology." http://www.fftw.org/speed/method.html. Accessed: 2015-12-13.

[12] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.