OpenCL Support for the OmpSs Programming Model

Vinoth Krishnan Elangovan § †, Rosa. M. Badia§ b, Eduard Ayguade Parra§ †

§ Barcelona Supercomputing Center † Universitat Politècnica de Catalunya,
bArtificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Spain

Abstract. The advent of heterogeneous computing has forced programmers to use platform specific programming paradigms in order to achieve maximum performance. This approach has a steep learning curve for programmers and also has detrimental influence on productivity and code re-usability. To help with this situation, OpenCL an open-source, parallel computing API for cross platform computations was conceived. OpenCL provides a homogeneous view of the computational resources (CPU and GPU) thereby enabling software portability across different platforms. Although OpenCL resolves software portability issues, the programming paradigm presents low programmability and additionally falls short in performance. In this paper we focus on integrating OpenCL framework with the OmpSs task based programming model using Nanos run time infrastructure to address these shortcomings. This would enable the programmer to skip cumbersome OpenCL constructs including compilation, kernel building, kernel argument setting and memory transfers, instead write a sequential program with annotated pragmas. Our proposal mainly focuses on how to exploit the best of the underlying hardware platform with greater ease in programming and to gain significant performance using the task parallelism offered by the OpenCL run time for multicore architectures. We have evaluated the platform with important benchmarks and have noticed substantial ease in programming with comparable performance.

1 Introduction

Microprocessor vendors have switched to the many core paradigm to effectively utilize the transistor count afforded by Moore’s Law. In accordance with this trend, we are seeing an increase in the number of cores with every successive product generation. It is also predicted that this trend is likely to continue into the future. The software development effort needed to harness the immense computing power, is however growing over the roof and hence presents the developers with a tedious challenge. In addition, with the emergence of heterogeneous computing models, it is imperative that expressive programming models be made available to the programmers to make proper use of the computational resource available and also ease the task of programming. Although lot of research is being carried out in this direction a clear solution to address this issue is still far
from sight. Emerging accelerator architectures address this issue by providing platform specific programming model. Two notable examples of this are GPUs from Nvidia (that use the CUDA programming model) [4] and the CELL processor from IBM (CELL programming API)[5]. Although these models provide the potential to get maximum performance out of the system (after exhaustive programming effort), the portability of applications developed using these models is largely limited. This restricts the application of this programming model to niche domains. Some believe that the trend of using platform specific programming models and development tools is here to continue.

An alternate approach to address the programmability issue involves design of platform independent paradigms in order to ease off the burden on the developers (by improving portability and code reuse). This approach has received a lot of attention lately. One such initiative currently being undertaken by the Khronos research consortium is OpenCL (Open Computing Language)[1]. OpenCL provides a platform independent programming APIs and is targeted towards developers to promote the concept of portability and reusability. The main drawback of this approach is that programs written using this model are cumbersome when compared to programs written using the platform specific model. This is primarily because the designers have traded-off programming efficiency for portability. In addition to the low programmability the development time involved here is quite overwhelming. In this paper we put forward a proposal to simplify the programming effort to develop applications using the portable programming model. The attempt is to integrate the OpenCL runtime with OmpSs, a Task Based programming model[2] and it as been productive with promising results. The pursuit is to offer the programmers a sequential programming flow with annotated pragmas specifying the key attributes for the code which is to be accelerated or the section which is needed to be parallelized for a target architecture. The OmpSs model comprises of Mercurium[2], a source to source compiler and Nanos runtime library[2] for effectively garnering the computing power of the hardware. The source code with pragmas is compiled with mercurium which links with the Nanos runtime hence forming a task-graph based on dependencies available for scheduling. This is scheduled in a appropriate way depending on the target architecture and uses the OpenCL runtime for executing the tasks. OmpSs, being a task-based programming model we focus our integration with the task parallel execution model of OpenCL(using a single compute unit). With this integration we try to offer the programmer a straightforward way to exploit heterogeneous architecture supporting code portability, reusability and minimal development time. In this paper we discuss the details of integrating OpenCL with OmpSs programming model and demonstrate how this proposed approach liberates the programmer from laborious development process.

The paper is divided into six sections. In the Section 2 we give a overview of the OpenCL programming API. Following that in Section 3, we discuss the OmpSs programming model developed at BSC and its key features. In section 4, integration of OpenCL with OmpSs is discussed in detail, giving insight to Nanos-OpenCL execution and memory model. Section 5 demonstrates our eval-
uation of the platform with key benchmarks. Section 6 concludes the paper and discusses possible extensions to this work.

2 OpenCL Overview

OpenCL (Open Computing Language), a Open standard parallel programming model introduced by Khronos to provide portability and code reusability across heterogeneous platforms (DSP Processors, CPUs and GPUs)[8]. It's a cross-platform programming language with a robust API capable of doing data parallel and task parallel computations across various architectures. It encloses a hierarchy comprising of the platform model, memory model, execution model and programming model (both data and task parallel)[8]. The design essentially is a classical host-client system with a host and others considered as OpenCL devices. The OpenCL devices are further divided into compute units and they are directed by the commands from the Host to do the computations. The computations executed on the compute units is fundamentally the portion of the application (kernel code) which needs to be accelerated.

This design involves several complicated steps as shown in figure 1 inorder to execute the kernel in any heterogeneous device. To start with, first the OpenCL platform is created and the device is identified and then corresponding context is created for the device with command queues. All the data transfers have to be accomplished with creation of OpenCL data buffers in the devices. The key aspect of portability here is the kernel code is being compiled and built at runtime [8] to create a executable for the corresponding device (GPU or CPU). Further it is required to set the corresponding kernel arguments for the kernel object created. This eventually makes the development more tedious and demanding leaving alone the performance optimization. Our contribution largely addresses this issue with OpenCL programming. The integration of OpenCL with OmpSs makes the development process a lot more simpler eventually writing a sequential

![OpenCL Programing model and OmpSs-OpenCL Integration Perspective](image-url)
program with added pragmas. OmpSs programming model developed at BSC is a combination of openMP and StarSs which is elaborated in the following section. Due to page constraints, we have provided reference links at appropriate places which covers the details of Language design and in depth explanation for the various concepts of OpenCL.

3 OmpSs-OpenCL Model

OmpSs is an adaptation of OpenMP with extensions based on the StarSs programming model[3]. It was designed to simplify programming for heterogeneous architectures using a unified development framework comprising the Mercurium compiler and the Nanos runtime. The OmpSs model currently encompasses the feature set provided by SMPSS[3], CellSs and GpuSs[7], each of which was developed keeping a specific architecture in mind. Since OpenCL offers cross platform portability and is starting to be recognized by multiple hardware vendors as a viable programming model for the future, we extend and integrate OmpSs with OpenCL to leverage on it. A brief perspective of the integration is shown in figure 1. The OmpSs-OpenCL platform follows the similar style of representation along with its previous feature sets. The parallel regions of the application are expressed in the form annotated pragmas which are considered to be Tasks by the model. The syntax of specifying the task includes the target device for execution and the necessary data required for it execution.

```
#pragma omp target device [clauses]
#pragma omp task [clauses]
```

The list of main clauses is the following:

- input ([list of parameters])
- output ([list of parameters])
- inout ([list of parameters])

The clauses to be specified for target device should be OpenCL Device (CPUs, GPUs) and for task is essentially the necessary data transfers as mentioned above. The clauses input, output and inout primarily express the datatype on which the task performs its computation along with size of data required. In addition their can be as many tasks as possible (eg: iterative task calls (same kernel code)) and as many number of task type (eg: multiple tasks with different kernel code). The task here is fundamentally the OpenCL task-parallel kernel, coded according to the OpenCL C99 standard[8] with the appropriate parameters matching with the task data clauses. Moreover the OpenCL kernel can be written in a separate file (for eg .cl file) and can be passed as a command line argument during compilation (multiple .cl files incase of multiple tasks). This annotated sequential program is compiled by mercurium and generates executable with corresponding calls to the Nanos runtime. Along with this mercurium also makes sure that the kernel code is passed to the runtime for compilation. From here the Nanos runtime using the OpenCL task parallel programming model tries to bring about the best possible parallel execution of tasks on the device. The
details of the integration with Nanos and mechanisms used for task parallelism are explained in the next section.

4 Nanos - OpenCL Model

Nanos is the asynchronous runtime environment used in OmpSs. It is based on a thread-pool execution model where the master thread coordinates and manages multiple slave threads. The executable generated by the compiler includes embedded calls that invoke different runtime services in Nanos for execution. Some of the key services provided by the Nanos runtime include task creation, dependency graph generation, memory transfer management and kernel execution management. Task creation service is responsible for creation and addition of task description to each of new tasks (typical example of descriptions include updating the target device, execution state, copy data information etc.). Following this, dependency graph generation service is responsible for generating the data dependency graph based on the task clauses specified in the application. Once the dependency graph is constructed, data dependencies are tracked and requests are sent to the software cache engine when appropriate to initiate the necessary transfers. Based on the dependency graph flow the task is set to be available for execution and is moved to the ready queue. The slave threads pick up tasks from the ready queue (on the basis of a specified scheduling algorithm) for execution. The integration is diagrammatically described in figure 3.

The baseline Nanos runtime environment supports three modes namely Performance, Debug and Instrumentation. Environment variables can be used to choose between any of these modes. Performance mode generally enables application execution in a performance optimal manner. Debug mode is generally used by developers to assist with the identification of issues like memory leaks. Instrumentation mode is used to generate detailed execution traces for further analysis and optimizations. Nanos is linked with instrumentation library paraver[6] in order to accomplish this. The Nanos-OpenCL model extends support to all the aforementioned modes (with additional support for trace generation to monitor OpenCL runtime activity). In in following sub sections we explain how Nanos environment is linked with the OpenCL runtime for doing memory transfers and execution of kernels.

4.1 Execution Model

The master thread as discussed previously is responsible for creating the tasks, generating dependency graph, scheduling tasks for execution and more importantly for creating the OpenCL runtime platform for Nanos. This happens immediately after the runtime is informed that the tasks are targeted towards an OpenCL device as shown in figure 3. Once the platform is created the device is identified and the OpenCL context is created for the corresponding device. The number of slave threads can be defined by the user based on application
requirements. Moreover the number of OpenCL command queues created correspond with the number of slave thread hence associating each slave thread with a OpenCL Command Queue. In addition to handling execution, the slave threads are responsible for compiling, building and argument setting by calling respective OpenCL calls. Work flow of slave thread is shown in figure 2. After the slave thread completes execution data is transferred back using OpenCL memory transfer calls initiated by the cache engine and the task state is changed to indicate completion. These data transfers are maintained by the cache engine which keeps detailed track of the inter task dependencies.

![Nanos Thread Execution Model](image)

**Fig. 2. Nanos Thread Execution Model**

### 4.2 Memory Model

Data dependencies specified using pragmas in the source code help Nanos to maintain a data dependency graph across the tasks. This service in Nanos is managed by the software based cache engine. When a task is created the copy over information is directly sent to the cache engine for each task inorder to maintain the data consistency. Once the task is available in the ready queue, it is ready to be scheduled for execution. The slave threads pick up tasks and send in calls to the cache engine to do the neccessary data transfers prior to slave threads executing tasks. The software based cache engine is integrated with the OpenCL runtime and performs allocation of buffers, data transfers using OpenCL runtime calls, for eg clcreatebuffer() to the device memory. This engine utilizes two different caching strategies: write back and write through. The user can choose any strategy based on the application requirement. Write back policy copies back the data from the device once the application has finished execution so that future tasks can reuse the data hence avoiding unnecessary data transfers whereas write through copies back the data once the task is over. Further the Cache engine also interacts with the slave threads to facililate the OpenCL call of clSetKernelArgs() for each task as shown in figure 3.
4.3 OpenCL Kernel Compilation

The key aspect of OpenCL is the runtime compilation of the kernel and this needs to be carefully handled by the Nanos runtime. With each task that is created it associates a parallel region of the code (kernel code). Since the compiler
passes the kernel code to the runtime, each slave thread when picking a task will eventually have the kernel code associated with it. This code is retrieved, compiled and build into a kernel object for execution. Incase of compilation failure Nanos throws the corresponding errors in the kernel code (task)to the user asking for debugging. Besides compilation it also tries to vectorize the kernel code for the device architecture. Once the kernel object is build successfully the slave thread contacts the cache system to set the arguments of the task and eventually enqueue the task for execution. Nanos makes sure that kernel code/task is compiled only once and if repeated or called iteratively, it uses the precompiled kernel hence bypassing compilation. This mechanism is maintained based on the unique kernel name based on which tasks are built as shown in figure 2. Moreover if multiple slave threads picks the same task(ideal kernel code) but with different data addresses (eg: tasks called iteratively with non blocking data) the runtime ensures that only a single thread compiles and the rest locks. After a single thread compiles and builds the other threads use the same kernel object but setting the appropriate kernel arguments for their respective tasks hence maintaining program correctness. Besides this strategy, parallel compilation among multiple slave threads is achieved when having different tasks with distinct kernel code are scheduled to them.

4.4 Device partitioning

By definition the task parallelism offered by OpenCL uses a single OpenCL thread and hence a single compute unit which corresponds to a single core in a multicore architecture. Using cl enqueueTask() for executing kernels uses only one core of the multicore machine making it a performance bottleneck. With task-parallel OpenCL behaving in this manner puts forth a hurdle to OmpSs inorder to utilize the machine to its maximum computing power. However, inorder to use all the cores of the system, each logical CPU cores of the system can be visualized into one physical processor which can be implemented using clCreateSubDevices () from OpenCL API extensions. This call can be used to partition the device into multiple sub-devices. This allows tasks to run parallel in different sub-devices and hence providing an opportunity to utilize them better. This feature is implemented in Nanos creating sub-devices equal to the number of slave threads instantiated. Each slave thread is associated to a command queue and with device partitioning feature we associate both to each sub-device. With this each slave thread enqueues tasks using its own command queue to its associated sub device. Consequently we try to use all the available cores of the device maintaining data synchrony among the different sub-devices. This is similar in spirit to concurrent kernel execution across the multiple cores in the machine. This scheme tries to use the all the available cores and hence making the runtime more scalable. In addition it also helps to keep the user away from the tedious process of creating subdevices and programming accordingly. We have not included evaluation of device partitioning in the paper as hardware vendors have not yet launched the latest OpenCL 1.2 package with partitioning facility. In Older OpenCL versions it is implemented as an extension and according to hardware vendors release notes[14] it is said to have a unstable behaviour.
Hence we have decided to wait for the next stable version for device partitioning experiments.

5 Evaluation

We evaluate our runtime system by analyzing different benchmarks with a OmpSs version against a standard OpenCL one. The OmpSs version follows sequential programming style with annotated pragmas targeting OpenCL device. This in turn uses the Nanos runtime for executing the parallel region (kernel code). We compare the execution time of OmpSs and with the original OpenCL task parallel version. We have found comparable performance with greater ease in programmability. The benchmarks were run with different problems sizes to check scalability and also experimented with data blocking strategy.

We have carried out our investigation in both GPUs and CPUs machines. Our CPU platform, Intel Xeon E7450 with 2.40GHz and NVIDIA Tesla M2090, the GPU platform. In the CPU we use OpenCL 1.1 however in GPU OpenCL version 1.0 is used. Mercurium and Nanos-OpenCL runtime were retained to be the same version for both the machines. Moreover, in both the systems Nanos was confined to performance mode so as to experience maximum optimizations in executing tasks.

5.1 Benchmarks

We choose to experiment the platform using three benchmarks from various computing domains. Typical double precision Dense matrix multiplication of two square matrices with varying problem size. Besides a normal matrix multiplication, a block partitioned matmul programmed in OmpSs were also considered for analysis. With growing use of GPUs in scientific computing we decided to use N-Body simulation as a benchmark for our comparison. Likewise Black scholes algorithm measuring option pricing from financial engineering is also taken for study. OmpSs version of Normal Matrix Multiplication is alike the OpenCL version with the kernel specified as a task. The execution of a single task(kernel) completes the multiplication. Whereas the blocked version, creates multiple tasks based on the block size with the initial tasks providing the data for latter tasks to work upon thereby benefitting on computations overlapped with communication. Black scholes iterates over the sample stock prices to find a optimum pricing. For each sample price a task is created and the kernel is run, calculating the option pricing using floating point operations. NBody simulation works with a large number of particles and in OmpSs version we create a task for each particle. Each Nbody particle runs the kernel thereby creating as many number of tasks as the particles. Figure 4 shows the graphical picture of the comparison of the 4 OmpSs benchmarks with original OpenCL version.

The execution timing graph of NBody simulation shows us with the increase in problem size, the OmpSs version slows down. This can justified as excessive amount of time is spent in task creation as a single task is created for each particle. Otherwise with nominal sizes OmpSs version offers comparable or sometimes better performance due to software cache management. The cache engine always
transfers the data even before the task is designated for its execution. Also when
the first task is compiled (kernel code) it is reused for all other particles with
only changing the arguments explained in figure 2.

![Graphs](image)

**Fig. 4.** Evaluation of OmpSs against OpenCL on CPUs and GPUs

The graph representing normal OmpSs matrix multiplication shows only neg-
ligible performance difference with OpenCL code. The OmpSs version uses the
same kernel as the OpenCL one and runs a single task providing comparable
performance with five fold decrease in lines of code. Interestingly, this can be
expressed in a partitioned manner benefitting the runtime environment. As we
can notice blocked Matrix Multiplication offers much better performance com-
pared to the normal OpenCL matmul version. This is predominately because
the tasks are created for specified blocked data, Moreover the computed data is
held in the device for other new tasks to consume, allowing the asynchronous
task parallel Nanos system to take advantage hence saving on data transfer time
which can be evidently seen with GPU results. This needs some effort from the programmer to understand the algorithm and partition accordingly and the remaining is well anchored by the Nanos runtime presenting better performance. In OmpSs version of BlackScholes is similar to OpenCL version wherein a single task is created for the kernel and all samples are computed. From the graph we can observe only minor differences in the execution timings. By considering the fact that data transfers to the GPUs are quite expensive, we can infer why GPUs have higher execution time in the evaluation. Moreover single compute unit of the GPUs is used since task-based OpenCL programming model is being experimented. From this evaluation we can say OmpSs-OpenCL integration offers satisfactory results with very less time spent on development. Additionally with partitioned data set and task parallel implementation of the application allows the asynchronous task parallel OmpSs Nanos environment to take benefit and can also outperform OpenCL execution time.

6 Related Work

With heterogeneous computing coming into mainstream research, several research groups and industry labs are involved in developing better programming models. OpenCL a open source programming API targeted at various architectures have inclined researchers to investigate deeper into the model aiming for better performance. In [10] a static partitioning of data parallel task are carried out for heterogeneous architecture. Based on prediction a portable partitioning scheme is proposed for dividing data parallel tasks for GPUs. Similarly [12] the data parallel tasks are dynamically scheduled to a CPU or GPU based on the contention state (metadata during runtime) and historical data (during profiling). In [11] talks about Hybrid OpenCL for distributed computing. The paper proposes the combined use of OpenCL and MPI for multiple node architecture. Also few translators such as [13] for converting CUDA to OpenCL have been proposed. Further in [9] supports OpenCL execution with a software cache for architectures with no cache(CELL processor). Summarizing, we find that researchers have restricted themselves to the data parallel OpenCL model and have neglected to address the laborious development time involved in it. Our approach mainly reduces the strenuous programming process in OpenCL targeting task-based OpenCL execution model.

7 Conclusion and Future Work

In this paper we propose an approach to integrate OmpSs and OpenCL programming models with an emphasis on reducing programmer effort and improving code portability and reusability. OpenCL programming model which is widely adopted as an industry standard provides a portable platform for programming heterogeneous architectures but falls short when it comes to programmability (Lot of effort is required). We present the integration of OpenCL with OmpSs programming model which can help eliminate the laborious programming process by empowering programmers with simple annotated pragmas. The approach is to use task parallel OpenCL model with asynchronous Nanos runtime system
and to take advantage of the hardware. We also discuss interactions between Nanos runtime environment and OpenCL in detail. Our experiments show comparable performance with programs written only using OpenCL thereby making a strong case for using OmpSs-OpenCL model.

We believe that current industry trends hold lot of promise for the proposed approach. With OpenCL 1.2 release supporting device partitioning, it would be possible to execute different tasks in each core of the machine and realize concurrent kernel execution in CPUs. In the GPU domain, product roadmaps predict GPUs with concurrent execution features by the end of the year thereby creating demand for robust programming models that offer good performance and reduces development effort. As part of future work we plan to extend the model to support multiple devices (both CPUs and GPUs) and run tasks on heterogeneous platforms. Moreover we also plan to tune the scheduler to suit OpenCL runtime flow to improve performance gains.

References

3. Josep M. Perez Rosa M. Badia Jesus Labarta “Handling task dependencies under strided and aliased references” Proceeding ICS ’10 Proceedings of the 24th ACM International Conference on Supercomputing
6. Parallel Program Visualization and Analysis Tool http://www.bsc.es/media/1364.pdf
11. Aoki, R. et al, Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. Parallel and Distributed Processing with Applications (ISPA), 2011