Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment

Vinoth Krishnan Elangovan[§ †], Rosa. M. Badia[§ ♭], Eduard Ayguadé [§ †]

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

Abstract. With heterogeneous computing becoming mainstream, researchers and software vendors have been trying to exploit the best of the underlying architectures like GPUs or CPUs to enhance performance. Parallel programming models play a crucial role in achieving this enhancement. One such model is OpenCL, a parallel computing API for cross platform computations targeting heterogeneous architectures. However, OpenCL is a low-level programming language, therefore it can be time consuming to directly develop OpenCL code. To address this shortcoming, OpenCL has been integrated with OmpSs, a task-based programming model to provide abstraction to the user thereby reducing programmer effort. OmpSs-OpenCL programming model deals with a single OpenCL device either a CPU or a GPU. In this paper, we upgrade OmpSs-OpenCL programming model by supporting parallel execution of tasks across multiple CPU-GPU heterogeneous platforms. We discuss the design of the programming model along with its asynchronous runtime system. We investigated scalability of four OmpSs-OpenCL benchmarks across 4 GPUs gaining speedup of up to 4x. Further, in order to achieve effective utilization of the computing resources, we present static and work-stealing scheduling techniques. We show results of parallel execution of applications using OmpSs-OpenCL model and use heterogeneous workloads to evaluate our scheduling techniques on a heterogeneous CPU-GPU platform.

1 Introduction

In the past decade, in order to deliver performance improvements, microprocessors vendors chose multi-core design paradigm to overcome memory, power and ILP walls. Today, multi-core CPUs package multiple homogeneous cores on a single die to increase data parallel computations. On the other hand, GPUs, with immense data parallel processing capability, are being exploited for general purpose computing (GPGPU) which traditionally handled graphics computations. Offering massive data parallel computing power, GPUs have become the focal point of today's High Performance Computing (HPC). Considering the recent progress of major chip manufacturers it is very clear that

in the future, laptops to HPC systems will consist of heterogeneous computing devices (CPU/GPU/DSP/FPGA). Thus presenting us with a hybrid/heterogeneous computing environment. This poses software developers with a significant challenge of best utilizing the underlying hardware. To harness this immense computing power for HPC, hardware vendors have built platform specific programming models like CUDA[16]. However, these models are quite demanding and involve significant software development time. Furthermore, these models suffer from portability issues, thereby pushing developers to rewrite code for different platforms from scratch. This makes heterogeneous programming quite tedious. In order to address this *state of the art* parallel programming, Khronos group[11][15] came up with OpenCL, an open source, platform independent, parallel computing API offering code portability. OpenCL solves the problem of using heterogeneous computing environment with a single programming model, but requiring significant programming effort for effective use. To address this issue, our previous work [8] proposes OmpSs-OpenCL programming model. It focuses on integrating OpenCL with OmpSs and discusses in detail the abstraction of OpenCL features and the semantics of OmpSs-OpenCL model. However, the model provides support either for a single GPU or a CPU. In this paper, we enhance OmpSs-OpenCL programming model to provide support for parallel execution of tasks in a CPU-GPU heterogeneous environment. Figure 1 depicts the architectural overview of the OmpSs-OpenCL programming model. The model includes the Mercurium compiler and the Nanos runtime system. The compiler does a source-to-source transformation of the user code written using OmpSs-OpenCL. During compilation the source code is embedded with appropriate Nanos runtime calls responsible for incorporating automatic data transfers and OpenCL execution. The runtime carries out task creation, task dependency analysis and scheduling. The Nanos software cache helps in tracking data locality of OmpSs-OpenCL tasks. This programming model supports all OpenCL compliant devices under a single operating system image, for instance, a system with 2 CPUs and 4 discrete GPUs can be programmed using OmpSs-OpenCL model with all 6 devices operating simultaneously.

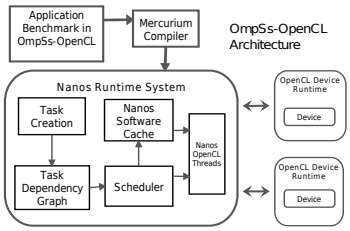

Fig. 1: OmpSs-OpenCL Programming Model - Architecture

With OmpSs-OpenCL supporting parallel execution of tasks across heterogeneous devices, there arises an opportunity to schedule these tasks efficiently across devices. In this paper, we discuss two scheduling methodologies namely static and work-stealing strategy. The static scheduling follows user specification of the target device for execution whereas the work-stealing essentially schedules tasks to the device that is devoid of work. The major contributions of this paper are the following:

- We enhance OmpSs-OpenCL model to support heterogeneous/hybrid environment supporting parallel execution of tasks.
- We present the design of the programming model for CPU-GPU systems and discuss its supporting runtime system.
- We discuss scalability and pre-fetching of OmpSs-OpenCL tasks and evaluate proposed scheduling strategies using heterogeneous workloads.

This paper is divided into seven sections. In the subsequent section, we describe the design of OmpSs-OpenCL programming model. In section III, we discuss the details of Mercurium compiler and Nanos runtime system for heterogeneous environments. Following that, in section IV, we present the scalability and pre-fetching of OmpSs-OpenCL tasks. In section V, we present static and work-stealing scheduling along with their evaluation using heterogeneous workloads. We give an overview of the related work in section VI and finally conclude with promising future extensions to this work in section VII.

2 OmpSs-OpenCL Model

OmpSs[7] is based on the OpenMP programming model with modifications to its execution model. It is primarily a task-based programming model focusing on abstraction of details to the user thereby making the programmer to write code in sequential flow with annotated pragmas for tasks(parallel regions). It uses a thread-pool execution model, where a master thread that starts the runtime and several other worker threads cooperate towards executing the tasks. Listing1.1 shows the directives supported by OmpSs-OpenCL model. These directives are used to annotate function declarations or its definitions. Each function annotated with task directives is considered an OmpSs-OpenCL task. The data environment of the task is obtained from its arguments. These arguments are specified with their directionality *input, output and inout* and its computing size. Using this information, dependencies across tasks are determined using StarSs dependency model[3]. Furthermore, *target device* clause is used to express heterogeneity, which can be *clcpu* and *clgpu*. *clcpu* undergoes OpenCL CPU execution and *clgpu* OpenCL GPU execution.

```
1  #pragma omp target device [clauses] NDRange(Parameters) [
       copy_type]
2  clauses:([clcpu][clgpu])
3  Parameters:Dimensions,GlobalGrid, LocalWorkGroupSize
4  copy-type : copy_in, copy_out, copy_inout, copy_deps
5  #pragma omp task [directionality]
6  Directionality
7  1.input ([list of parameters])
8  2.output ([list of parameters])
9  3.inout ([list of parameters])
10 #pragma omp taskwait
```

Listing 1.1: Directives supported by OmpSs-OpenCL

The *copy_in*, *copy_out* and *copy_inout* clauses are used to specify where the data have to be available, produced and both. *copy_deps* clause is used to specify that if the task has any dependence clauses, then they will also have copy semantics. The task-wait construct (Listing1.1-line 10) can be used to introduce a barrier after parallel code. Along with these directives, NDRange will accept OpenCL execution configuration for the particular task/kernel. These parameters essentially represent the OpenCL grid dimensionality[11], *NDRange* Global Grid and the *LocalWorkGroupSize*[11]. Moreover, in OmpSs-OpenCL, the task definition is essentially an OpenCL kernel written according to OpenCL C99 standard by the user. The task/kernel code can be defined in the same file of the source or in a separate .cl file. The following sample gives a comprehensive understanding of how to use the OmpSs-OpenCL directives(Listing 1.2).

```
1  #pragma omp target device (clcpu) ndrange(1,0,size,512)
       copy_deps
2  #pragma omp task in ([size]a) out([size]c)
3  void copy_task(double *a, double *c, int size);
4
5  #pragma omp target device (clgpu) ndrange(1,0,size,512)
       copy_deps
6  #pragma omp task in ([size]c) out([size]b)
7  void scale_task (double *c, double *b, int size);
8
9  #pragma omp target device (clgpu) ndrange(1,0,size,512)
       copy_deps
10 #pragma omp task in ([size]a,[size]b) out([size]c)
11 void add_task (double *a, double *b, double *c, int size)
       ;
12
13 #pragma omp target device (clgpu) ndrange(1,0,size,512)
       copy_deps
14 #pragma omp task in ([size]b,[size]c) out([size]a)
15 void triad_task (double *b, double *c, double *a, int
       size);
16
17 int main(int argc, char** argv)
18 {
19     copy_task(a,c,size);
20     scale_task (c,b,size);
21     add_task(a, b,c,size);
22     triad_task (b, c, a,size);
23     #pragma omp taskwait
24 }
```

Listing 1.2: OmpSs-OpenCL Example Program

Listing 1.2 is an example code in OmpSs-OpenCL for heterogeneous environment. As shown, the directives are used to specify on which device the task should execute (*clcpu or clgpu*). The directive *task* mentions the required data by the task in lines 2,6,10 and 14. Also, *target device* for the task is mentioned

in lines 1,5,9 and 13. When the tasks are invoked, the runtime checks for data dependencies between them, if independent, they can be executed in parallel in the CPU-GPU environment. Listing-1.2 example program is a stream application [14]. The definition of the tasks is essentially a OpenCL kernel. The kernel code for *triad_task* is shown in listing 1.3.

```
1  #pragma OPENCL EXTENSION  cl_khr_fp64 : enable
2  __kernel void triad_task ( __global double *a, __global
       double *b, __global double *c, __const double scalar,
       __const int size)
3  {
4     int j=get_global_id(0);
5     a[j] = b[j]+scalar*c[j];
6  }
```

Listing 1.3: OmpSs-OpenCL Task/Kernel

As per specification, copy_task (line 19) is the only task in the benchmark undergoing OpenCL CPU execution and the remaining are executed in the OpenCL GPU. We can apprehend from this example that the user can avoid tedious OpenCL API calls and also realize parallel execution across different devices, leaving users to write the task/kernel code alone. The next section gives a detailed description of the Nanos runtime.

3 OmpSs-OpenCL Execution Model

In this section, we discuss the implementation details of OmpSs-OpenCL programming model for hybrid environments. The model embodies Mercurium compiler and the Nanos runtime system.

3.1 Mercurium Compiler

Mercurium compiler [4] is a source-to-source compilation infrastructure supporting C and C++. It checks syntax and semantic errors for annotated pragmas in the program and parses the OmpSs-OpenCL source generating the associated runtime calls for the parallel regions/tasks to be executed. These runtime calls embodies information about the directionality of the data transfers in order to deduce input/output dependencies among all tasks in the source. With inclusion of *clcpu or clgpu* as *target device*, necessary changes to the compiler have been carried out to generate their respective OpenCL-Nanos runtime calls. When Mercurium encounters a task declaration or invocation which is targeted for *clcpu or clgpu* it generates corresponding task creation runtime call with the respective .cl file as a parameter to it. The task/kernel code written according to OpenCL standard is left untouched by the compiler as it undergoes target device specific runtime compilation. Mercurium is only responsible to generate appropriate calls to the runtime and check the correctness of user code. The back-end of the model involving Nanos runtime does the significant portion of the work in order to experience parallel execution of tasks across multiple OpenCL devices.

3.2 Nanos Runtime Environment

The Nanos runtime library is the backbone of OmpSs-OpenCL programming model. It is a task-based runtime system with asynchronous execution flow. It creates an acyclic task dependency graph based on the *task* directive information. The runtime carries out the services including task creation, task dependency graph generation, data transfers, synchronization between tasks and execution of tasks. The Nanos master thread is responsible for most of these services whereas the worker threads are accountable for task execution. Each Nanos worker thread corresponds to a device in the heterogeneous environment and work in parallel. In the following subsections we discuss the enhancement implemented in the runtime to support hybrid OpenCL environments. Figure 2 presents OmpSs-OpenCL model diagrammatically.

Support for Heterogeneity Nanos runtime essentially invokes a master thread which in turn controls the worker threads. These worker threads (Nanos-*arch* thread) implements the functionalities for different architectures. For OmpSs-OpenCL model the Nanos-OpenCL worker thread implements OpenCL execution for a CPU or GPU device[8]. However, for hybrid environments the equation changes. Although, OpenCL API is open-source but the implementation of its runtime varies with every vendor (eg., Nvidia, Intel, AMD, ARM ..). Hence, not all devices can use the same OpenCL package. We need to have vendor-specific OpenCL package in order to use different devices. To tackle this issue we implement two different OpenCL-Nanos architectures (Nanos-OpenCL-CPU thread and Nanos-OpenCL-GPU thread), to realize both CPU and GPU OpenCL behavior respectively. The Nanos-*arch*-OpenCL Thread is responsible for compilation, argument setting and execution of the task/kernel. With CPU and GPU OpenCL architecture module, OmpSs-OpenCL supports all OpenCL compliant CPU and GPU systems. Figure 2 shows the Nanos-OpenCL-*device* thread model linking with the corresponding OpenCL package.

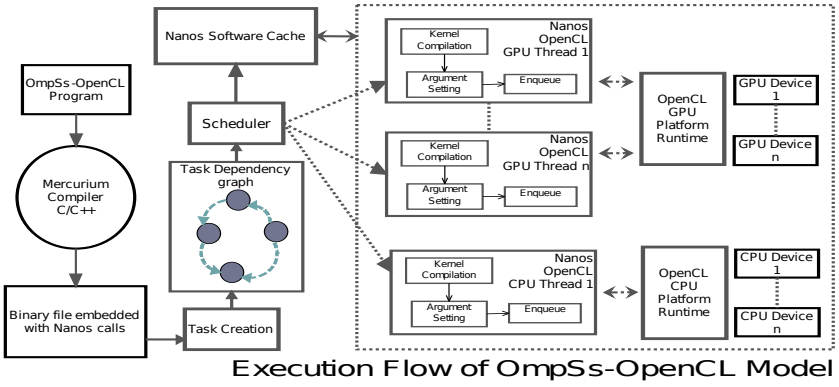

Fig. 2: Execution Flow of OmpSs-OpenCL

Nanos-OpenCL Thread Model The Nanos-OpenCL thread is built over the OpenCL runtime. This thread is responsible for receiving the task for execution from the scheduler. The thread carries out task compilation, kernel argument setting and enqueue for execution (figure 2). During kernel enqueue, the runtime uses the *NDRange dimensions* (kernel launch parameters) from the user-defined NDRange clause of the task and launches it using *clEnqueueNDRangeKernel* [11]. In addition, each OpenCL device is associated with a Nanos-OpenCL thread (eg., 2 Nanos-OpenCL-GPU threads for 2 GPUs available). This approach is employed to provide parallel services to multiple devices under the same platform. Furthermore, if the same task/kernel code using different data is scheduled to each of the 2 devices in a platform for execution, (Eg: 2 GPUs using Nvidia's OpenCL runtime) the runtime makes sure that the task is not compiled twice for the same platform. Moreover, each thread operates on its own OpenCL *CommandQueue*[11] specific to its device.

Nanos-OpenCL Memory Model The Nanos Software Cache manages data transfers in and out of devices. The software cache is linked with both CPU and GPU OpenCL packages in order to carry out data transfers respectively. Once a task is scheduled to its targeted device the scheduler informs the cache engine with required information to initiate respective transfers for the task into the device. The cache system uses the task dependency information from the runtime in order to initiate transfers based on the locality of its required data.

The Nanos software cache incorporates data locality principles into its transfer management. In general, the software cache system tries to minimize the total number of transfers from memory to its devices and vice-versa. In listing 1.2, consider stream benchmark with 4 different tasks all targeted towards GPU execution exhibits different data dependence across one another. With all the tasks targeted for GPU execution, total number of transfers required for application execution is 10, 6 input and 4 output transfers according to its *#pragma* specification. If locality of each data source is considered before transfers, the number of transfers can be brought down to 4(1 input and 3 output). To elaborate, *copy_task* input, *a* vector is transfered as input and its output *c* is not transferred back as it provides the input for dependent *scale_task*. Similarly, the other 2 tasks also receive its data from its predecessor. The output transfers for *scale_task*, *add_task* and *triad_task* are carried out for *b, c, a* vectors respectively. Figure 10 shows the difference in total data transfer timing for both CPU and GPU with and without data locality for two different problem sizes of stream benchmark. With an average of more than 50% reduction in the transfer time can be understood from Figure 10. We agree it is not possible for the cache system to optimize data transfer timing of this order for all applications but we can guarantee that it cannot cause any overhead. With the runtime system orchestrating data transfers and execution of the tasks, the synchronization needed to maintain data flow and program correctness are also incorporated within it. With GPUs spending more time in data transfers due the PCI bottleneck[9], it is very crucial to hide transfer latency in order to achieve better performance. We realize this by incorporating task-prefetching technique into our runtime. In

this technique, runtime does not wait for the executing task to finish but communicates with the scheduler to prefetch the next available task. When ready tasks are available for scheduling with no data dependencies with the executing task, the Nanos-OpenCL thread picks it thereby initiating the cache system to perform the required data transfers into the device. Such transfers overlap with the execution of the current task achieving communication-computation overlap. Also, the prefetcher only fetches data if it can fit it in the available OpenCL allocation space, otherwise it does not prefetch. With Nanos-OpenCL threads working in parallel, all available devices including CPUs and GPUs can prefetch in parallel. In the next section, we investigate on scalability of OmpSs-OpenCL tasks on multiple GPUs with task-prefetching.

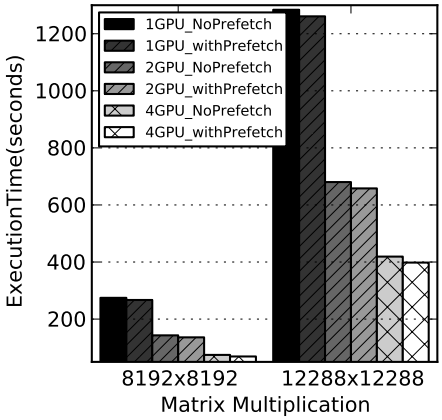

Fig. 3: Matrix Multiplication

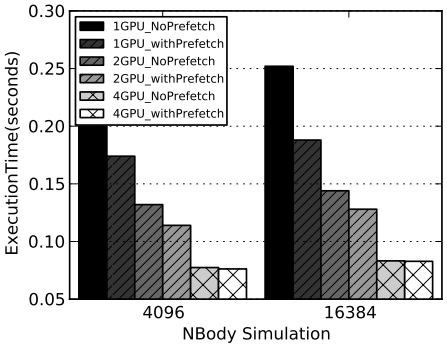

Fig. 4: NBody Simulation

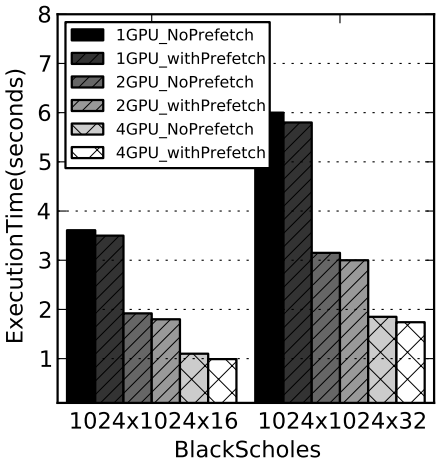

Fig. 5: BlackScholes

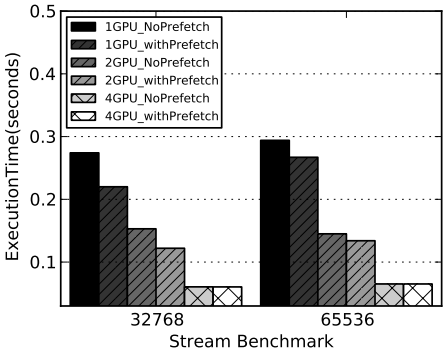

Fig. 6: Stream Benchmark

4 Scalability on Multiple GPUs

We experimented with 4 benchmarks namely, Matrix multiplication, Nbody simulation, Stream and Blackscholes with two different problem sizes for scalability on four Nvidia Tesla C2070 GPUs each having 448 CUDA cores with CUDA compiler driver V5.5. The four benchmarks are written in OmpSs-OpenCL with computations partitioned into parallel tasks in order to exploit the available 4 GPUs. 12288x12288 Matmul is partitioned into 27 tasks with each task computing 4096x4096(blocksize) and 8192x8192 into 64 tasks with each task computing on 2048x2048 block matrix size. BlackScholes is partitioned into 64 tasks and Nbody into 4 tasks. All three benchmarks are compute bound whereas Stream, a memory bandwidth limiting benchmark consist of 4 tasks in itself(add,copy,scale and triad). Original OpenCL benchmarks were obtained from [1][5]. With the runtime facilitating prefetching of tasks, our experiments also demonstrate the benefit in overlapping computation with communication. Figures 3, 4, 5 and 6 show the scalability of benchmarks on four GPUs with and without task-prefetching. All 4 benchmarks experience good scalability with an average speedup of 4x compared to its execution on a single GPU.

Using task-prefetching, matrix multiplication experiences notable gain in performance. With both problem sizes matmul gains an average of 5% in performance. To illustrate, for 12288x12288 matmul with 27 tasks, 80% of the total time taken for transferring input data is overlapped with the computation when benchmarked using 4 GPUs. The data transfers of the first 4 tasks getting scheduled to the 4 GPUs cannot be overlapped but the rest of 23 task inputs are prefetched with effective realization of computation-communication overlap for matmul. Table 1 depicts in detail percentages of overlapping experienced in both matmul and blackscholes for 4 GPUs. In Stream and Nbody benchmarks, prefetching with 4 GPUs does not provide improvements as it is partitioned into 4 tasks only. On an average, using task-prefetch there is 5% gain with matmul

Application for 4 GPUs	Total Transfer time	Overlapped	%Overlapped
Matmul 12288x12288 (27 Tasks)	24.8 sec	19.6 sec	80%
Matmul 8192x8192 (64 tasks)	7.2 sec	6.6 sec	90%
Black Scholes 16 (64 tasks)	134 ms	125 ms	93%
Black Scholes 32 (64 tasks)	200 ms	184 ms	92%

Table 1: Task Pre-Fetch Gain

and BlackScholes, 15% with NBody and 10% with Stream Benchmark compared to experiments without prefetching using 1,2 and 4 GPUs repectively. With this evaluation, we show OmpSs-OpenCL support for multiple GPUs with parallel execution of tasks across them realizing good scalability.

5 Scheduling

With OmpSs-OpenCL supporting parallel execution of tasks on hybrid CPU-GPU environment there arises a chance to schedule them efficiently in order to use the resources in the best possible way. Code portability in OpenCL allows kernels to run on any OpenCL compliant platform. The reason for having the OmpSs-OpenCL task as a pure OpenCL kernel is to make it adaptive, so that it can be scheduled to any other device (other than the *targetdevice*) based on machine availability. We present two different scheduling techniques for OmpSs-OpenCL model described in the following subsections.

5.1 Static Scheduling (S)

Static scheduling is a straightforward scheduling mechanism wherein the task is scheduled to the device mentioned using the *target device* clause. In case of application programmer having incorporated device-specific optimizations into the task/kernel, this approach would be implicit to go forward with. Applications which can be partitioned into tasks suited specifically to CPUs or GPUs using optimizations like vectorization, device specific *workGroupSize*, total number of *workGroups*, loop unrolling and local memory optimizations are advised to be statically scheduled. In these scenarios, static scheduling would bring out the best performance from their corresponding devices.

5.2 Work-Stealing Scheduling (WS)

The concept of work-stealing is essentially executing a task with clause *target device (clcpu)* on a GPU system and vice-versa. In this mode the scheduler assigns the task which is ready for execution to the first Nanos-OpenCL thread which goes idle. When the goal is to utilize all available devices in the environment, work-stealing scheduling would be very pertinent. However, work-stealing strategy can degrade performance, if the data needed by a task has to be transferred across devices for execution. The scheduler checks with the Nanos software cache engine for data locality before the task is scheduled. For example: given that a OpenCL-CPU is idle, if task B consumes output data from task A being executed on a GPU, the scheduler assigns task B to the GPU in order to save time from unwanted data transfer(GPU→Host→CPU). So dependent tasks are scheduled to the same device ensuring data locality, hence, preserving the application from a unwanted data transfer.

5.3 Evaluation

We evaluate both static and work-stealing techniques using MinoTauro supercomputer, a multi-GPU system running Linux with two Intel Xeon E5649 6-Core at 2.53 GHz and two NVIDIA GPUs M2090 with 512 CUDA cores. To the best of our knowledge, benchmark suites for hybrid environments composing CPUs

and multiple discrete GPUs are yet to be available. Thus, for our evaluation, we have developed heterogeneous workload with 8 different benchmarks written using OmpSs-OpenCL model. Original OpenCL benchmarks are obtained from [1][5].

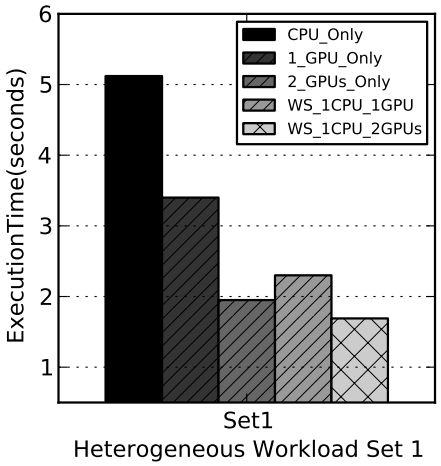

Fig. 7: Workload Set 1

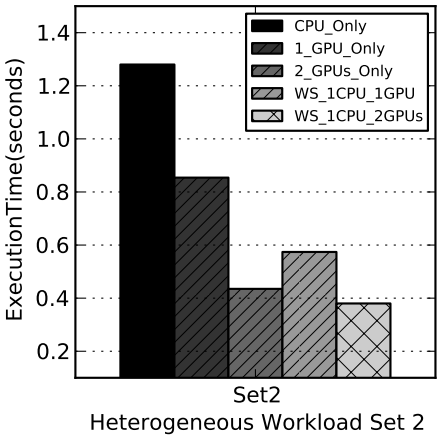

Fig. 8: Workload Set 2

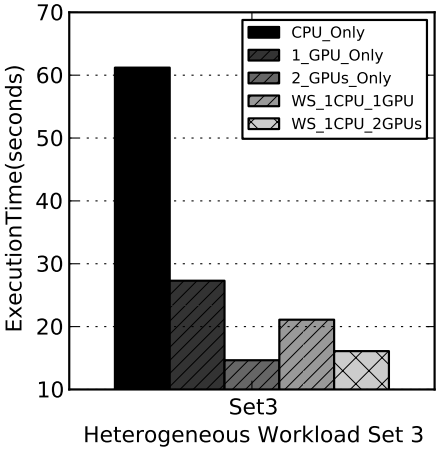

Fig. 9: Workload Set 3

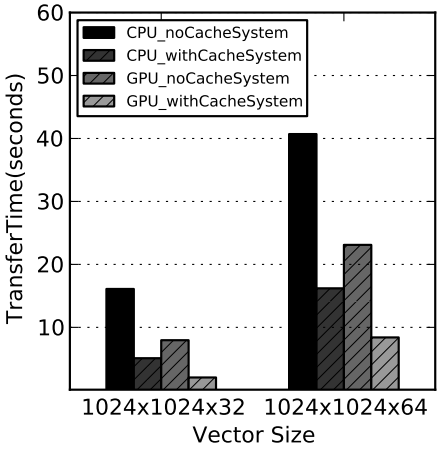

Fig. 10: Nanos Cache System

For our experiments, we create 3 heterogeneous workloads namely, set 1,2,3 computing with different data-sizes. All benchmarks in the workload are independent from one another, providing possible parallel execution of tasks allowing us to investigate static and work-stealing techniques. In particular, Blocked mat-

mul(8 independent tasks), Bitonic sort(independent tasks based on problem size) and Stream(4 dependent tasks) are partitioned, whereas other benchmarks including Blackscholes, Reduction, Nbody simulation, Convolution and Transpose compute using only a single task. Figure 7, 8 and 9 shows the evaluation of the three workload sets on our heterogeneous platform, with task-prefetch facility activated. Table 2 gives the overview of the workload characterization with its problem size, execution timings on both devices and scheduling decisions taken using work-stealing technique.

Benchmark	Set	Problem Size	CPU Execution Time(ms)	GPU Execution Time(ms)	Sch Decision(WS-1Gpu)	Sch Decision(WS-2Gpus)
Blocked Matmul	1	1024x1024	259.75	109.1	Cpu and Gpu	Cpu and Gpu
	2	512x512	27.4	9.1	Cpu and Gpu	Cpu and Gpu
	3	2048x2048	8123.3	3367.6	Cpu and Gpu	Cpu and Gpu
Matrix Transpose	1	2048x2048	5.2	1.0	Gpu	Gpu
	2	1024x1024	2.1	0.257	Gpu	Gpu
	3	512x512	1.24	0.0714	*Cpu*	Gpu
Black Scholes	1	1024	0.0622	0.009	Gpu	Gpu
	2	4096	0.138	0.11	Gpu	Gpu
	3	16384	0.215	0.0170	Gpu	Gpu
Bitonic Sort	1	4096	0.089	0.008	Cpu and Gpu	Cpu and Gpu
	2	512	0.0377	0.016	Cpu and Gpu	Cpu and Gpu
	3	1024	0.034	0.012	Cpu and Gpu	Cpu and Gpu
Convolution	1	256	0.582	0.0071	Gpu	Gpu
	2	1024	0.88	0.040	*Cpu*	Gpu
	3	4096	1.4	0.0544	Gpu	Gpu
NBody	1	4096	22.8	6.02	Gpu	Gpu
	2	512	0.569	0.715	Gpu	Gpu
	3	1024	2.89	1.42	Gpu	Gpu
Stream	1	1024	0.12	0.028	Gpu	Gpu
	2	4096	0.18	0.024	Gpu	Gpu
	3	8192	0.310	0.0412	Gpu	Gpu
Reduction	1	4096	0.571	0.010	*Cpu*	Gpu
	2	1024	0.488	0.010	Gpu	Gpu
	3	16384	1.01	0.0121	*Cpu*	Gpu

Table 2: Heterogeneous Workload Characterization

Set 1 with benchmarks using both larger and smaller problem sizes, work-stealing with both 1 and 2 GPUs provides performance gain for the workload. In our evaluation (Figure 7, 8 and 9), the performance gain calculated is in comparison with the execution time of workloads using static schedule with 1

and 2 GPUs. Using WS [1] 1CPU-1GPU experiences almost 30% performance gain with Reduction benchmark running on the CPU and both Blocked matmul and Bitonic sort with multiple tasks gets shared across both devices. In WS 1CPU-2GPUs the gain is 12% with Blocked matmul and Bitonic sort executed among both devices. Set 2 workload consist of benchmarks with small sizes provides gain of 33% and 10% for 1CPU-1GPU and 1CPU-2GPU repectively. Convolution gets scheduled to CPU with WS 1CPU-1GPU and both bitonic and blocked matmul gets scheduled to both devices in both work-stealing cases.

Set 3 workload consist of benchmarks working with larger data-sets. WS with 1CPU-1GPU provides 22% performance gain with Reduction and transpose undergoing CPU execution. However, WS with 1CPU-2GPUs experiences performance loss of 13%(Figure 9). This is to say static schedule of Set 3 to 2 GPUs perform better compared to WS with 1CPU and 2 GPUs. In this case, adoption of WS technique is not favorable on the performance front although it utilizes all available devices. Apparently, tasks inherently suited for GPU can be scheduled to CPU. In this case, Blocked matmul exhibits significant difference in execution time with CPU execution being much slower compared to GPU and is scheduled to both GPU and CPU accounts for this loss in performance. Moreover, with 2 powerful GPUs in WS mode enhances tasks to be scheduled to them compared to a single available CPU thereby making device count and its characteristics a crucial parameter to be considered during scheduling.

From Table 2, we can learn that all benchmarks for different problem sizes work faster in the GPUs. Currently, considering the standard that GPUs are not standalone systems and CPUs are inherent in heterogeneous systems, utilizing CPUs effectively becomes crucial. With GPUs and CPUs becoming more powerful and power efficient and with every generation, it is imperative for the user to use both devices in a hybrid environment as HPC components. Moreover, with supercomputing design going heterogeneous, the design of algorithms and workloads involving mixture of varied computations exhibiting different types of parallelism would be critical. This puts forth a need to have a single programming model to harness these resources. OmpSs-OpenCL model provides this facility with complete abstraction to the user programmer. Further, in order to effectively use the resources we plan to extend our work focusing on development of an optimal scheduling methodology. With extensive analysis of our workloads, we would like to characterize parameters like computational complexity of tasks, data transfer timings, runtime status, device load/contention [10] which comprehensively define the best possible device for scheduling based on the execution environment.

6 Related Work

With GPUs becoming prominent in HPC domain, lots of research groups have focussed on their programmability. [13], SnuCL provides OpenCL framework extending original OpenCL semantics for heterogeneous cluster environment. This

[1] WS-Work-Stealing Scheduling Technique

framework does not offer an abstraction to the users to make OpenCL programming easier unlike OmpSs-OpenCL. In [2], StarPU run-time environment offers programming language extensions supporting task-based model. StarPU model expects user to know OpenCL API calls (eg: Kernel launch and argument setting) in order to write applications using them. Whereas, OmpSs-OpenCL provides complete abstraction to OpenCL API for programming heterogeneous systems. CAPS HMPP [6] is a toolkit with a set of compiler directives, tools and runtime supporting parallel programming in C and Fortran. It is based on hand-written codelets defining the parallel regions to be run on accelerators. OpenACC [12] high level programming API describes a collection of compiler directives used to specify regions of code for offloading from a host CPU to an attached accelerator but compared to OpenCL it is not yet adopted among all microprocessor and accelerator vendors. In [17], the author describe a wrapper model *clUtil* which simplifies OpenCL API. It provides a wrapper to OpenCL, where the user can skip OpenCL constructs by replacing appropriate clUtil calls in order to do its functionalities. Whereas, OmpSs-OpenCL model offers users to write simple sequential style programming with no complex calls to the runtime. In general, OmpSs-OpenCL programming model offers a high-level uniform programming approach. With current supercomputers already following hybrid designs, we believe that OmpSs-OpenCL model can be very effective in realizing productive supercomputing.

7 Conclusion and Future Work

In this paper, we upgrade OmpSs-OpenCL programming model to support parallel execution of tasks across CPU-GPU hybrid systems. We discuss the implementation of Nanos runtime system which plays a key role in realizing heterogeneous computing. Along with this support, we present static and work-stealing scheduling techniques. Static scheduling is user-assisted scheduling, whereas work-stealing utilizes all the available devices in the system, in turn increasing its throughput. We evaluated scalability of 4 benchmarks using OmpSs-OpenCL model on 4 GPUs providing an average speedup of 4x compared to one GPU. We used three sets of heterogeneous workloads to investigate parallel execution of tasks on hybrid platform using both static and work-stealing strategies. With OmpSs-OpenCL model being extensible, we look forward to support heterogeneous devices like Intel Xeon-Phi and FPGAs. Moreover, we plan to use the results from our evaluation to devise an optimal dynamic scheduling algorithm for CPU-GPU hybrid systems.

8 Acknowledgement

We thankfully acknowledge the support of the European Commission through the TERAFLUX project (FP7-249013) and the HiPEAC-2 Network of Excellence (FP7/ICT 217068),the support of the Spanish Ministry of Education (TIN-2007-60625, TIN-2012-34557, CSD2007-00050 and FI program) and the Generalitat de Catalunya (2009-SGR-980).

References

1. AMD. Amd sdk examples. *http://developer.amd.com/tools-and-sdks/heterogeneous-computing/*.
2. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
3. Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Euro-Par 2009 Parallel Processing*, pages 851–862. Springer, 2009.
4. Barcelona Supercomputing Center. The nanos group site: The mercurium compiler. *URL http://nanos. ac. upc. edu/mcxx*.
5. Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.
6. Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.
7. Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
8. Vinoth Krishnan Elangovan, Rosa M Badia, and Eduard Ayguade Parra. Ompss-opencl programming model for heterogeneous systems. In *Languages and Compilers for Parallel Computing*, pages 96–111. Springer, 2013.
9. Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.
10. Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. Opencl task partitioning in the presence of gpu contention.
11. Khronos OpenCL Working Group et al. The opencl specification. *A. Munshi, Ed*, 2008.
12. OpenACC Working Group et al. The openacc application programming interface, 2011.
13. Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.
14. John D McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.
15. Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.
16. CUDA Nvidia. Programming guide, 2008.
17. R Webber. clutil-making opencl as easy to use as cuda(website).