Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices

Prasanna Pandit
Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India
prasanna@hpc.serc.iisc.ernet.in

R. Govindarajan
Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India
govind@serc.iisc.ernet.in

ABSTRACT

Programming heterogeneous computing systems with Graphics Processing Units (GPU) and multi-core CPUs in them is complex and time-consuming. OpenCL has emerged as an attractive programming framework for heterogeneous systems. But utilizing multiple devices in OpenCL is a challenge because it requires the programmer to explicitly map data and computation to each device. The problem becomes even more complex if the same OpenCL kernel has to be executed synergistically using multiple devices, as the relative execution time of the kernel on different devices can vary significantly, making it difficult to determine the work partitioning across these devices a priori. Also, after each kernel execution, a coherent version of the data needs to be established.

In this work, we present FluidiCL, an OpenCL runtime that takes a program written for a single device and uses both the CPU and the GPU to execute it. Since we consider a setup with devices having discrete address spaces, our solution ensures that execution of OpenCL work-groups on devices is adjusted by taking into account the overheads for data management. The data transfers and data merging needed to ensure coherence are handled transparently without requiring any effort from the programmer. FluidiCL also does not require prior training or profiling and is completely portable across different machines. Across a set of diverse benchmarks having multiple kernels, our runtime shows a geomean speedup of nearly 64% over a high-end GPU and 88% over a 4-core CPU. In all benchmarks, performance of our runtime comes to within 13% of the best of the two devices.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming; C.1.4 [**Processor Architectures**]: Parallel Architectures; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – Single-instruction-stream, multiple-data-stream processors (SIMD)

CGO '14, February 15 - 19 2014, Orlando, FL, USA

ACM 978-1-4503-2670-4/14/02$15.00.
http://dx.doi.org/10.1145/2544137.2544163

General Terms

Performance

Keywords

FluidiCL, OpenCL, Runtime, Heterogeneous Devices, GPGPU

1. INTRODUCTION

Graphics Processing Units (GPU) have become commonplace among a range of computing devices, from mobile phones to supercomputers [21]. While originally meant purely for graphics purposes, over the past few years, they have become very popular in the scientific community for running general purpose data-parallel applications. This has resulted in computing systems becoming "heterogeneous" - with multi-core CPUs and GPUs, with different capabilities being available for programmers to exploit.

OpenCL [12] has emerged as a popular programming model for heterogeneous computing, with nearly all major vendors of GPU and CPU hardware providing runtime software to enable OpenCL programs to run on them. GPU execution has shown tremendous performance for several applications [15], but there remain other classes of applications, especially those with irregular data accesses or control flow in them that likely run better on multi-core CPUs [19]. OpenCL allows the programmer to explicitly specify the device on which a kernel can be run. However, this requires the programmer to know whether the CPU or the GPU is more suited for the given kernel. This task of choosing the right device for each kernel is completely left to the programmer, making performance tuning difficult. Further, the relative performance of a program may be input dependent, owing to the overheads of data transfer between devices. Picking the right device for the workload, or using all devices together therefore remains a challenge. The problem is exacerbated if the application has several kernels, each of which runs faster on a different device.

Apart from this, it is also not possible to use multiple devices synergistically to execute a single kernel without resorting to large investment of time and effort from the programmer, since this involves not only work assignment but also data management across devices having discrete memory. This places further burden on the programmer to effectively use all the devices available to him.

To address this problem, ideally, it is required to have a single runtime system that takes an OpenCL program written for one device, identifies the appropriate work partitioning of the kernel on CPU and GPU, and synergistically executes the kernel on both the CPU and the GPU, maintains consistent view of (shared) data across devices and provides the best possible performance. It is also desirable that such a runtime not impose the need for prior profiling or training before any application is run. Not only would

this improve existing programs, it would also unlock performance potential of other programs which hitherto yielded meagre benefits on the GPU, and in the process encourage more programs to be ported to OpenCL and make it a truly heterogeneous computing platform.

Our solution to the problem is to use a dynamic work distribution scheme that attempts to run each OpenCL kernel synergistically on both devices and automatically performs data transfers and merging to ensure data coherence. We have designed and implemented a runtime for heterogeneous devices that does not require prior training or profiling and is completely portable across different machines. Because it is dynamic, the runtime is also able to adapt to system load. The runtime factors in data transfer overheads, and causes the kernel execution to "flow" towards the faster device. We refer to our runtime system as FluidiCL (pronounced as "fluidical") as it allows a fluidic movement of workload across different devices. FluidiCL can achieve fine-grained work partitioning across multiple devices, performing the required data management automatically.

We have experimented with a set of benchmarks on a system with an NVidia Tesla C2070 GPU and a quad-core Intel Xeon W3550 CPU. Our benchmark results show that FluidiCL is able to pick the best of two devices for all applications, giving an overall geomean speedup of 64% over the GPU and 88% over the 4-core CPU with the best improvement being $2.24\times$ over the better of the two. We have compared the performance of FluidiCL to SOCL [25], the StarPU [1] OpenCL extension and find that FluidiCL outperforms SOCL by $2.67\times$ with StarPU's default scheduler. FluidiCL also does $1.26\times$ better when the recommended dmda scheduler is used without requiring any calibration or profiling steps of SOCL.

2. BACKGROUND

In this section, we give a brief overview of the OpenCL programming standard and introduce some basic terminology. OpenCL is an open standard of computing for heterogeneous devices developed by Khronos Group [12]. Vendors who support OpenCL for their devices ship OpenCL runtime software and compilation tools which facilitate development of OpenCL programs for their devices. Currently OpenCL is supported by most major CPU and GPU vendors including AMD, Intel, NVidia, ARM and Apple [10]. OpenCL allows runtime software from different vendors to coexist and allows programs to utilize multiple platforms in the same program. Figure 1 shows different OpenCL vendor runtimes for different devices.

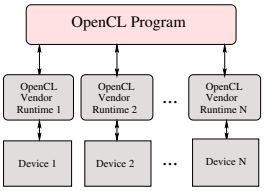

Figure 1: Different OpenCL devices and runtimes

OpenCL programs consist of the host program which runs on the CPU and coordinates activities with the runtime using API functions and kernel programs which run on the *compute device*. Most OpenCL programs are written in a way that initializes the platform and devices (`clCreatePlatform`, `clCreateDevice`), creates buffers for data on the device (`clCreateBuffer`), transfers data into the buffers (`clEnqueueWriteBuffer`) and launches the kernel program on the device (`clEnqueueNDRangeKernel`).

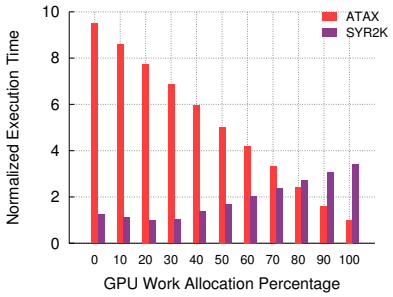

Figure 2: Comparison of ATAX and SYR2K

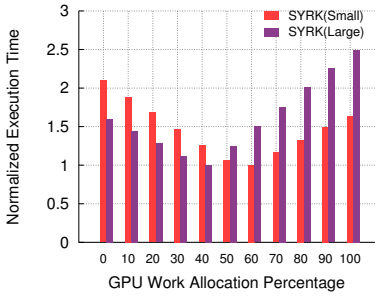

Figure 3: Comparison of SYRK for two input sizes

The host program transfers the results back to the host after kernel completion (`clEnqueueReadBuffer`). OpenCL provides command queues for enqueuing the above data transfer and kernel execution commands.

On the device, execution of kernel programs are carried out by *work-items*, which execute the same program but act on different data and are allowed to follow different control flow paths. A group of these work-items together form a *work-group*. The index space of work-items is called an NDRange which can be one, two or three-dimensional. Work-items and work-groups are identified by an N-tuple in an N-dimensional NDRange. The corresponding CUDA terminology is *thread* for a work-item, *thread block* for a work-group and *grid* for an NDRange.

OpenCL uses a relaxed memory consistency model that guarantees that memory is consistent across work-items of a work-group at barrier synchronizations (`barrier`), but not between work-items from different work-groups [11].

3. MOTIVATION

In this section, we motivate the need for adaptive work distribution and synergistic execution of OpenCL programs by showing that it is possible to get better performance by using both the CPU and the GPU than one device, and that the amount of work to be assigned to a device differs both by application and by input size. Also, we demonstrate a case where different kernels within an application prefer different devices and argue that decisions taken at a kernel level may not result in the best application performance.

We consider two applications ATAX and SYR2K from the Polybench [5] suite run on our experimental system with an NVIDIA Tesla C2070 GPU and an Intel Xeon W3550 CPU (see Section 8 for details). The graph in Figure 2 plots the normalized execution time of the kernel when the percentage of work allocated to the GPU for the two applications is varied. From the speedup observed from different work assignments, it can be seen that ATAX shows

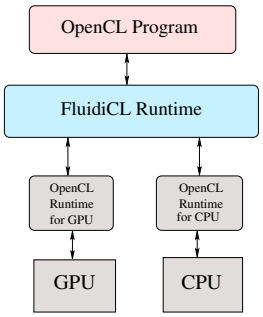

Figure 4: OpenCL Runtime System showing FluidiCL

maximum performance when run on GPU alone, whereas SYR2K benefits most when 20% of the work is run on the GPU and the rest on the CPU. This means that, in the absence of an automatic work distribution framework, every application programmer would need to spend considerable time and effort in order to identify the best performing work partitioning across devices.

Manual work distribution done statically may still not guarantee good performance. Figure 3 demonstrates this using the benchmark SYRK from the Polybench suite. The graph is plotted with different input sizes of SYRK – one with the smaller input size of $(1100, 1100)$ and the other with the larger size of $(2048, 2048)$. From the graph, it can be observed that the best performing workload assignment varies with different input sizes. A split of $60/40$ between the GPU and the CPU works well with the smaller input while for the larger input, a work distribution of $40/60$ works best.

The problem becomes even more complex for applications with more than one kernel, where work allocation decisions for each kernel made statically may not work well. We illustrate this with the help of BICG from the Polybench suite. As shown in Table 1, this application has two kernels, each of which runs faster on a different device. To make use of the better performing device, there is a need to do data management and the proposed scheme should take this into account. Otherwise, the cost of data management could negate all benefits and could well lead to an overall poorer performance than if the entire application had been run on one device.

Kernel Name	CPU Only	GPU Only
BICGKernel1	0.179	0.379
BICGKernel2	1.766	0.022

Table 1: Kernel Running Times in seconds for BICG.

To overcome these challenges, we design and implement an OpenCL runtime for heterogeneous systems called FluidiCL which adapts to different application and input characteristics and executes each kernel across the CPU and the GPU. It also takes into account data transfer overheads and tracks data location in order to support OpenCL programs with multiple kernels effectively.

4. THE FLUIDICL RUNTIME

The FluidiCL runtime is designed as a software layer that sits on top of existing vendor runtimes and manages the devices using their OpenCL API as shown in Figure 4. FluidiCL exposes a set of API functions identical to the OpenCL API to the application. The application invokes them as it would in any single-device OpenCL program. For each API function, our runtime invokes the appropriate vendor function of one or both the devices as required.

For the purpose of work-distribution, we use an OpenCL work-group as a unit of allocation. This is because according to the OpenCL memory consistency model [11], each work-group can execute independently of the other and the results computed by a work-group are guaranteed to be visible to other work-groups only at the end of the kernel execution[1]. We use *flattened* work-group IDs while assigning work. A flattened work-group ID is a one-dimensional numbering of work-groups for multi-dimensional NDRanges. An example of flattened work-group IDs for a two-dimensional NDRange is shown in Figure 5. In this figure, a total of 25 work-groups are laid across 5 rows and 5 columns. The flattened work-group ID is also specified in this figure in boldface letters.

(4,0) **20**	(4,1) **21**	(4,2) **22**	(4,3) **23**	(4,4) **24**
(3,0) **15**	(3,1) **16**	(3,2) **17**	(3,3) **18**	(3,4) **19**
(2,0) **10**	(2,1) **11**	(2,2) **12**	(2,3) **13**	(2,4) **14**
(1,0) **5**	(1,1) **6**	(1,2) **7**	(1,3) **8**	(1,4) **9**
(0,0) **0**	(0,1) **1**	(0,2) **2**	(0,3) **3**	(0,4) **4**

Figure 5: Flattened Work-Group Numbering. Each work-group ID is shown as a tuple (x,y) with x and y being the row and column IDs. The number below shown in boldface is the flattened ID of the work-group.

Figure 6 gives an overview of heterogeneous kernel execution done by FluidiCL. The diagram shows that kernels are being launched on both the GPU and the CPU. First the required data transfer to both devices is initiated using data transfer calls. On the GPU, the entire NDRange is launched, with a work-item in every work-group performing checks regarding the execution status on the CPU. The CPU kernel launches are broken up into what we call "subkernels" which execute only a small chunk of the NDRange at a time, starting from the highest flattened work-group ID to the lowest. The work-groups in the subkernels can be executed by the available CPU cores concurrently. Thus while the GPU is executing the work-groups from the lower end, the CPU cores are executing the work-groups from the upper end. The CPU communicates the data and status to the GPU at the end of each subkernel. When the GPU reaches a work-group that has already been executed by the CPU (detected by checking the completion status sent by the CPU) the GPU kernel stops executing the work-group and the kernel is said to have been completed. Next the data merge step happens on the GPU, which combines both the CPU and the GPU computed data. This is followed by a device-to-host transfer to bring the final data to the CPU. As an example, the figure shows a work-group completion status for a kernel with N work-groups, with the CPU executing 50 work-groups in each subkernel. Whenever the GPU finds that the remaining work-groups have been executed on the CPU, the kernel execution ends and data merge takes place.

We now describe our runtime in detail.

4.1 Platform and Buffer Setup

FluidiCL sets up OpenCL platforms and devices for both the CPU and the GPU as part of its initialization. The call for kernel compilation (`clBuildProgram`) results in kernel compilation being done for both devices. When the user program requests for buffers to be created using the `clCreateBuffer` function, our runtime creates buffers for both the CPU and the GPU. Every host-to-device data transfer command done using `clEnqueueWrite-`

[1]In the absence of atomic primitives, which we do not support.

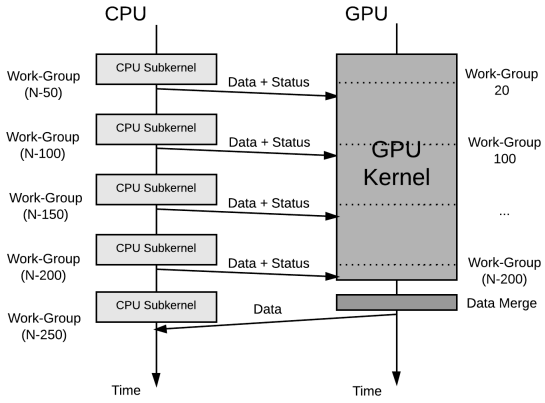

Figure 6: Overview of Kernel Execution

`Buffer` is translated into two commands – `clEnqueueWrite-Buffer (gpu, //...)` and `clEnqueueWriteBuffer -(cpu, //...)` transfers to both devices. Before a kernel is executed, for every buffer that is going to be modified by the kernel (*out* or *inout* variables – which can be identified using simple compiler analysis at the whole variable level), additional buffers are created on the GPU to hold the intermediate data coming in from the CPU and to maintain a copy of the unmodified data. These buffers are used for merging the data computed by the devices after kernel completion (as explained later in Section 4.3).

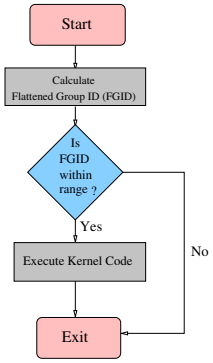

Figure 7: CPU Kernel Execution Flowchart

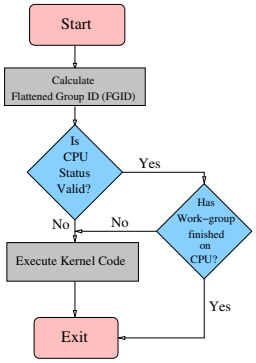

Figure 8: GPU Kernel Execution Flowchart

4.2 Kernel Execution

When a kernel is enqueued on the GPU, FluidiCL enqueues the kernel as before on the GPU. In addition, a small fraction of the workload is enqueued on the CPU for execution by means of a kernel call to the OpenCL CPU runtime. We call this a CPU subkernel. The work-groups are assigned to the CPU in the decreasing order of flattened work-group IDs, because the GPU would start executing from the other end. Thus both devices (CPU and GPU) would be executing non-overlapping parts of the NDRange. A CPU scheduler thread runs on the host that monitors the state of the subkernel execution and assigns more work to the CPU cores as and when it gets completed.

At the end of each subkernel, the CPU transfers the computed data (out and inout data) and follows it up with an execution status message to the GPU. The GPU kernel periodically checks the status of the CPU execution in the first work-item of every work-group and aborts execution of a work-group if it finds that the work-group has already been completed on the CPU (as illustrated in the flowchart shown in Figure 8). A data-merge step follows on the GPU which puts together the results computed on the two devices. The details of the data merging are presented in Section 4.3. Note that since the execution status from the CPU always follows the computed data using an in-order command queue, the GPU considers a work-group complete only when the computed results are already with it and ready for merging. This is a crucial part of our work since it automatically takes into account data transfer overheads before any work is considered complete on the CPU. Thus our runtime takes into account not only the execution time in the other device, but also the data transfer time. Whenever the computed data for a work-group has not arrived at the GPU yet (even though the execution may be complete), the GPU kernel takes over the execution of the work-group and computes it as it would have anyway.

In kernel execution where CPU execution of subkernels along with the associated data transfer can be overlapped with GPU execution, both CPU cores and the GPU execute the kernel synergistically. In kernels where the CPU performance lags the GPU so much that it is not useful at all, no new status message arrives at the GPU and the GPU kernel executes the entire NDRange. On the other hand, if the CPU is able to compute the entire NDRange first, then the results of the GPU execution are ignored and the final data is deemed to be available on the CPU. This way, the faster device always does more work and we ensure that in kernels where execution on both devices does not help, we still do as well as the faster of the two devices.

Figures 7 and 8 show the CPU and GPU kernel execution flowcharts respectively. On the CPU, every work-group checks if the its ID is within the range specified by the subkernel argument. If it is, then the kernel code is executed. On the GPU, every work-group checks if the CPU has already executed the work-group by querying the CPU execution status variable available with it. If it has not, then the work-group is executed. We describe kernel implementation in more detail in Section 5.

4.3 Data Merging

At the end of a kernel execution, the partial buffers computed by the CPU and the GPU need to be merged. This merging is carried out by maintaining a copy of the unmodified buffer (the *original* buffer) and comparing it with the data sent by the CPU (the *diff* step) and copy the CPU computed data if it differs from the *original* (the *merge* step). A simple code snippet in Figure 9 shows how this is done.

As the merging step is completely data parallel, it is carried out on the GPU. In our implementation, we use the size of the base data type of the buffer pointer to decide the granularity of the *diff* and *merge*. For example, for a four byte integer, four bytes of data are compared and copied. This type information is stored during the GPU kernel execution as a metadata at the beginning of each buffer. Figure 9 shows data merging at the granularity of one byte for illustrative purpose.

4.4 Device-to-Host Transfers

After the kernel execution and data merging, the final data is brought back to the CPU, so that subsequent kernels may execute on the CPU as well. At the end of the data transfer, the buffer version number (as explained later in Section 5.3) and location information for the buffer are updated. Note that these data transfers

```
1  __kernel void md_merge_kernel(__global char * cpu_buf,
2    __global char *gpu_buf, __global char * orig,
3    int number_bytes)
4  {
5      // cpu_buf contains data computed by the CPU. orig
6      // contains a copy of unmodified data before either
7      // devices started.
8
9      // index gets the work-item ID. The number of work-items
10     // launched depends on the size of the data to be merged.
11     index = get_global_id (0);
12     if (index < number_bytes &&
13             cpu_buf[index] != orig[index])
14     {
15         gpu_buf[index] = cpu_buf[index];
16     }
17 }
```

Figure 9: Data Merging on GPU

are only done for buffers which are modified in that kernel (*out* or *inout* variables). For kernels that have executed completely on the CPU, this data transfer is not necessary and is not performed.

5. IMPLEMENTATION

FluidiCL has been implemented in C with the Pthreads library used for launching additional threads for performing scheduling on the CPU and for doing device-to-host data transfers. Each API in the OpenCL program is replaced with the corresponding FluidiCL API, with no change in arguments. This is done for each application with the help of a simple find-and-replace script. The CPU and GPU kernels are modified by adding the required checks as described in the flowcharts in Figure 8 and Figure 7. The GPU kernel code contains an abort check at the beginning of every work-item. The CPU kernel code contains a check to ensure that the launched work-group falls in the desired range of this subkernel. In the current implementation, the kernel transformations have been done manually. But these are simple transformations that can be automated using a source-to-source compiler.

When the `clEnqueueNDRangeKernel` function is called, the corresponding FluidiCL function creates the necessary buffer copies required for data merging (Section 4.3). The GPU kernel is then enqueued for launch. A thread is spawned on the host using the Pthread library which repeatedly launches subkernels on the CPU until all the work-groups have finished execution. This is achieved by monitoring whether the GPU kernel has exited. The CPU scheduler thread may also exit if all the work-groups have finished execution on the CPU. After every CPU subkernel execution, the CPU scheduler thread makes a copy of the output buffers and enqueues them for sending to the GPU, allowing more CPU subkernels to be launched concurrently.

5.1 Adaptive Chunk Size allocation

In the FluidiCL runtime, the number of work-groups to be allocated to the CPU is decided dynamically. The initial *chunk size* is set to 2% of the total work-groups. We want to be able to send the computed data and status as frequently as possible to the GPU, but having too small a chunk size may result in too many CPU kernel launches and associated overheads, and might result in poor utilization of the compute units on the CPU. Therefore, we use a heuristic to adaptively compute the number of work-groups to be launched in each subkernel. Each subkernel run is timed and the average time per work-group is computed. FluidiCL keeps increasing the chunk size so long as the average time per work-group keeps decreasing. We use this heuristic because we observe that increasing the number of work-groups per subkernel leads to improvement in time taken per work-group and this trend continues till there are sufficient number of work-groups for the OpenCL runtime to utilize the CPU effectively. We have determined experimentally that setting the initial chunk size to 2% and increasing it in steps of 2% of total work-groups works well across all benchmarks. If the number of work-groups to be launched is less than the number of available compute units on the CPU, we set the chunk size to the available number of compute units to ensure full resource utilization. Later, in Section 6.3, we describe an optimization that ensures we keep all compute units busy by splitting work-groups into smaller chunks.

5.2 Offset Calculation

Before a CPU subkernel is launched, the work-groups to be executed have to be identified. This is done by keeping track of the flattened work-group ID of the last work-group launched so far, and subtracting the number of work-groups to be executed in this subkernel (CPU subkernels launch work-groups in the decreasing order). The number of work-groups to be executed is determined by the adaptive chunk size selection (Section 5.1). Once the start and end flattened work-group IDs are determined, they are converted into actual work-group IDs in *N* dimensions, *N* being the number of dimensions in the host kernel launch parameter. The NDRange dimensions for the CPU subkernel are then determined along with appropriate work-group offsets. In the case of *N* being greater than 1, the CPU subkernel launches a NDRange slice with more work-groups than needed, and passes the flattened work-group IDs of the start and end work-groups as parameters to the subkernel. The subkernel on the device then skips the execution of the work-groups outside the specified range. This is done to enable the execution of arbitrarily shaped NDRanges in each subkernel by utilizing the offset parameter option of OpenCL kernel launches.

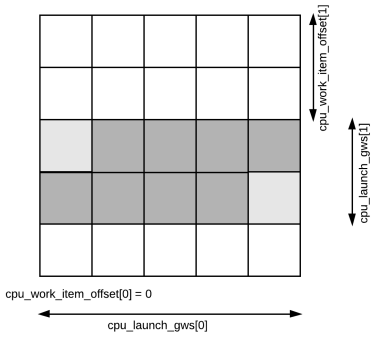

Figure 10: Offset calculation in a two-dimensional NDRange

For example, consider a two-dimensional NDRange as shown in Figure 10. If the work-groups shown in dark gray are desired to executed on the CPU, the CPU scheduler launches all the shaded work-groups, and passes the flattened work-group IDs of the start and end work-groups to be executed on the CPU. In the figure, `cpu_work_item_offset` is the offset for the CPU subkernel launch and `cpu_launch_gws` is the number of work-groups launched in each dimension of the CPU subkernel NDRange.

5.3 Buffer Version Tracking

Every time a kernel execution is requested, the kernel is assigned a kernel ID by FluidiCL. All the CPU-to-GPU and GPU-to-CPU data transfers initiated by FluidiCL use this kernel ID as version numbers in order to identify the kernel which wrote to them. The runtime maintains for each buffer the expected and received buffer versions. At the beginning of a kernel all the *out* and *inout* buffers have their expected buffer version set to the kernel ID. The received buffer versions are set to this same kernel ID at the end of the kernel execution, either when data is received back at the host by the D2H thread or if the CPU has executed the entire NDRange. These buffer versions are used to check if all input buffers needed by a subsequent kernel launch are available with the CPU before allowing CPU kernel to proceed. If the CPU data is stale i.e., the buffer versions with the corresponding kernel ID have not yet arrived, the CPU scheduler thread waits until the most recent data arrives from the GPU before starting execution. However, the GPU kernel is free to proceed while the CPU waits for its data to arrive, as the GPU has the most up-to-date data version. The FluidiCL design also uses buffer versions to ensure correctness by discarding stale data if messages arrive late to the GPU.

5.4 Additional Command Queues

FluidiCL creates additional command queues for the GPU in order to facilitate host-to-device and device-to-host transfers independent of the application command queue. The two queues, called *h2d* and *d2h* queues are created as part of the FluidiCL runtime initialization. These queues respectively hold the commands for transferring CPU computed data to the GPU and the merged results on the GPU to the CPU. These command queues, along with additional buffer copies, help in overlapping communication with kernel execution, as described in Section 5.5.

5.5 Overlapping Computation and Communication

FluidiCL carefully overlaps all data transfers with computation on both the CPU and the GPU. Intermediate copies of data are made on the CPU to allow subsequent CPU subkernels to proceed while the data is transmitted to the GPU. For D2H transfers, a copy of the *out* buffers are made to their respective *original* buffers at the end of the kernel execution so that subsequent kernel launches may be allowed to proceed on the GPU while the results of the previous kernel are returned to the host in parallel. These *original* buffers can be used during data merging of the next kernel.

5.6 Device-to-Host Thread

When the CPU scheduler thread determines that no more work needs to be scheduled for a kernel, it launches a device-to-host thread by using a Pthread call. This thread initiates a buffer read from the GPU for every *out* or *inout* buffer and transfers each one of them to their respective CPU buffers. Once it has determined that the transfers have completed, the received buffer version numbers are set to the appropriate kernel ID (Section 5.3).

6. OPTIMIZATIONS

In this section, we describe optimizations for reducing overheads on the host and to improve performance of the OpenCL kernels.

6.1 Buffer Management

As described in Section 4, FluidiCL needs to create additional buffers on the GPU. These buffers are used to hold additional copies of the buffer – one for data coming in from the CPU and the other to maintain the original buffer. To avoid the creation and destruction overheads of these buffers for every kernel, we maintain a pool of available buffers on the GPU which are reused repeatedly. Every time a new CPU-copy or original buffer is requested, a buffer from the common pool is returned. If the size of the requested buffer is more than any of the available buffers, a new one is created and returned. A flag is set for every buffer in use. At the end of each kernel execution, older unused buffers are freed and GPU memory is reclaimed.

6.2 Data Location Tracking

Along with the buffer version tracking, FluidiCL also maintains information about where the most recent version of the data can be found. When the user requests data using `clEnqueueReadBuffer` call, this location information helps us avoid making a data transfer from the GPU in case it is already available on the CPU. This could be either as a result of D2H transfers already being done automatically, or because of the CPU executing all the work-groups.

6.3 CPU Work-group Splitting

Since only a small fraction of the work-groups are assigned to the CPU at a time, we need to ensure that all the available hardware threads are utilized. Since the AMD CPU OpenCL runtime that we use in our work runs each work-group as a single thread with the work-items being executed in a loop, any work allocation for the CPU that is less than the number of hardware threads would under-utilize the device. One way to overcome this problem is to allocate as many work-groups as there are compute units (hardware threads). But if the NDRange consists of a small number of long running work-groups, then it may be more beneficial to split each work-group and run it in parallel, rather than running one or more of them that may not finish in time.

We perform an optimization for the CPU kernel that we call *CPU Work-group Splitting*. We split a single work-group into as many work-groups as there are available compute units. Since one work-group is being executed as multiple work-groups, we replace the `barrier` function with a custom helper function implemented by us to ensure correctness. Also, local work-group ID and work-group size are replaced appropriately. Since `__local` data is visible only within a work-group, all local memory buffers are replaced by global memory buffers. Since CPUs do not contain dedicated hardware for `__local` data, using `__global` for them does not hurt performance.

6.4 GPU Work-group Abort in Loops

As explained in Section 4.2, the work-items in modified GPU kernels contain code to abort the execution of a work-group if it has already been executed on the CPU. If the kernel contains loops, for effective termination of work-groups, this abort code has to be placed inside the innermost loops also. Otherwise it leads to duplication of work on both the devices and inhibits potential gains. Note that it is legal to partially write data to the output GPU buffers, because the data merging step that follows a kernel overwrites this partial output by the data computed by the CPU.

6.5 Loop Unrolling for the GPU

If the GPU kernel abort checks are done inside loops, it may inhibit loop unrolling done by the GPU kernel compiler in some cases. This can degrade GPU kernel performance especially in cases where the loop body is very short. Therefore we manually add loop unroll code after abort checks done inside loops. Figures 11 and 12 show an example of how unrolling is done.

```
1  int  md_flattened_blkid  = mdGetFlattenedBlockID (
2           get_local_size  (0));
3  __local  bool  donotexecute;
4
5  //Kernel abort check at the beginning of the kernel.
6  checkCPUExecution (*cpu_exec_status,  kernel_id ,
7           md_flattened_blkid ,  &completedAtCPU);
8
9  if  (completedAtCPU == true)
10     return;
11
12 int  k;
13 for (k=0;  k< m; k++)
14 {
15     //Some expression  involving  k
16 }
```

Figure 11: Before loop unrolling

```
1  int  md_flattened_blkid  = mdGetFlattenedBlockID (
2           get_local_size  (0));
3  __local  bool  completedAtCPU;
4
5  //Kernel abort check at the beginning of the kernel.
6  checkCPUExecution (*cpu_exec_status,  kernel_id ,
7           md_flattened_blkid ,  &completedAtCPU);
8
9  if  (completedAtCPU == true)
10     return;
11
12 for ( int  k=0;  k< m; k+=(1+UNROLL_FACTOR))
13 {
14     //Kernel abort check done inside loops also.
15     checkCPUExecution (*cpu_exec_status,
16              kernel_id ,  md_flattened_blkid ,  &completedAtCPU);
17
18     if  (completedAtCPU == true)
19         return;
20
21     int  k1,  ur  = UNROLL_FACTOR;
22     if  ((k+ur)  > m)
23         ur  = (m-1)-k;
24     for  (k1 = k;  k1 <= (k+ur);  k1++)
25     {
26         //Expression  where k  is  replaced  with k1
27     }
28 }
```

Figure 12: After loop unrolling

6.6 Online Profiling and Optimization for CPU kernels

The OpenCL kernel functions in the application generally contain optimizations specific to the device it is intended to run on. Optimizations done for improving GPU performance would likely run poorly on the CPU. For example, kernels which contain optimizations for improving memory coalescing would result in poor cache locality on the CPU [22]. For this reason, we let the user or an optimizing compiler provide more than one implementation of a kernel if desired, each of which would contain device specific optimizations. Given these alternate kernel versions, FluidiCL automatically performs online profiling and picks the best version of the kernels. It starts by running each kernel version for a small allocation size and measuring time taken for its completion. The best performing kernel version is picked for the remaining subkernels. Currently we restrict all kernel versions considered for online profiling to have the same arguments and be functionally identical in terms of output buffers modified.

7. LIMITATIONS OF FLUIDICL

Currently any OpenCL program which does not use atomic primitives in its kernels can use FluidiCL to execute on both the CPU and the GPU. Also, kernel execution calls to FluidiCL are blocking in our current implementation. FluidiCL implements the most commonly used subset of OpenCL API for reading and writing buffers (`clEnqueueReadBuffer` and `clEnqueueWriteBuffer`) and therefore applications that use other API need to be rewritten to use these supported function calls. Since FluidiCL aims to accelerate each kernel by running it on multiple devices, long running kernels with high compute-to-communication ratio benefit more from it than applications with a large number of short kernels.

FluidiCL has been implemented for accelerators on a single node and therefore, is not designed to support multiple devices across different nodes. However, it can use all the CPU cores, or multiple CPUs on different sockets, since the underlying OpenCL CPU runtime supports them. It can also support other accelerators like Intel Xeon Phi as long as they are present in the same node.

8. EXPERIMENTAL METHODOLOGY

We evaluate the FluidiCL runtime on a machine having an NVidia Tesla C2070 GPU and a quad-core Intel Xeon W3550 CPU, with hyper-threading enabled. Our experiments use all the eight available CPU hardware threads. We use the OpenCL runtime from the AMD APP SDK 2.7 for the CPU. We use the AMD runtime because we found it almost consistently better performing than Intel's OpenCL runtime. For the GPU, we use the OpenCL runtime shipped with NVidia CUDA SDK 4.2. We use six benchmarks from the Polybench [5] suite which contain kernels that run sufficiently long for our purposes. The benchmarks show varying characteristics in terms of their running time on the two devices, the number of kernels in them and the number of work-groups. Details of the benchmarks we used along with input sizes, number of kernels and number of work-groups are shown in Table 2.

For the baseline CPU-only and GPU-only results, we run each benchmark using the vendor runtimes directly. For both the baseline and the FluidiCL results, we run each benchmark ten times and collect total running time for each benchmark (which includes all data transfer overheads) and take their average. We exclude the results from the first run to avoid runtime kernel compilation overheads showing up, as the vendor OpenCL runtimes seem to cache and reuse kernel compilation results. We believe this is a fair methodology. We also exclude the OpenCL platform initialization time from all results, which takes a small fixed amount of time. In this work, identification of *out* buffers, kernel code modifications to add the abort checks and optimizations like loop unrolling and CPU work-group split are done manually. However these are very simple in nature and can easily be done by a source-to-source compiler.

Benchmark	Input Size	Kernels	Work-groups
ATAX	(28672, 28672)	2	896, 896
BICG	(24576, 24576)	2	96, 96
CORR	(2048, 2048)	4	8, 8, 16384, 8
GESUMMV	(20480)	1	80
SYR2K	(1000, 1000)	1	4000
SYRK	(2500, 2500)	1	24727

Table 2: Benchmarks used in this work. All benchmarks are from the Polybench suite [5].

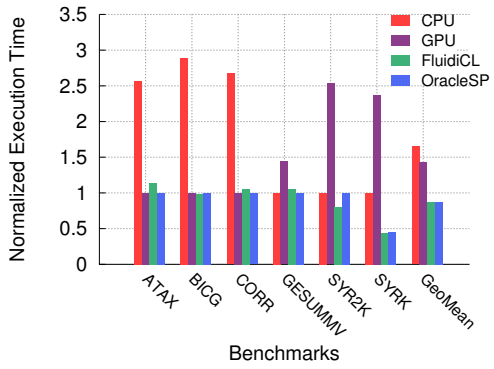

Figure 13: Overall performance of FluidiCL. For each benchmark, execution time is shown normalized to the best performing single device – CPU-only or GPU-only.

9. RESULTS

In this section, we show the performance of the benchmarks when run with the FluidiCL runtime and compare it with the CPU-only and GPU-only execution. We then demonstrate the effectiveness of our runtime when presented with different inputs. Also, we show the effects of optimizations described in Section 6 on each of the benchmarks. Finally, we compare our approach with the StarPU [1] OpenCL extension (SOCL) [25] in Section 9.4.

9.1 Overall Performance of FluidiCL

First we present the results of running each benchmark for the input sizes shown in Table 2. All applications have been run with all optimizations enabled except the online profiling optimization described in Section 6.6. Figure 13 plots the total running time of each benchmark normalized to the faster of the CPU-only and the GPU-only executions. In the figure, the rightmost bar represents oracle Static Partitioning (OracleSP) performance. OracleSP has been obtained by running each benchmark with different CPU/GPU work allocations with $x\%$ to CPU and $(100-x)\%$ to GPU, with x incremented from 0 to 100 in steps of 10% in each run. OracleSP reports the result of the best performing split determined statically. From the figure it can be seen that FluidiCL consistently does as well as the best of the two devices and outperforms it in three benchmarks – BICG, SYR2K and SYRK. Also, FluidiCL achieves a performance comparable to OracleSP in all benchmarks, except ATAX where it is within 14%. ATAX is slower than OracleSP because the benchmark performs best when run on GPU alone, which both OracleSP and FluidiCL do; but the one-time cost of creating additional buffers on the devices causes the slight performance degradation for FluidiCL. Further, in SYR2K and SYRK, FluidiCL outperforms OracleSP. This is because OracleSP is essentially a static partitioning scheme which cannot adapt to finer grained work allocations which might have worked well for this benchmark whereas FluidiCL is able to automatically choose these fine-grained partitioning at runtime depending on the workload and how well the application performs on the individual devices. Overall, FluidiCL outperforms GPU-only by $1.64\times$, CPU-only by $1.88\times$ and the best of the two by $1.14\times$. The speedup in case of SYRK is greater than $2\times$ due to improved GPU cache performance caused by FluidiCL's modified kernel code.

9.2 Different Input Sizes

Next we demonstrate that FluidiCL can adapt very well to different input sizes. Figure 14 shows this using the SYRK benchmark when run with inputs from $(1000, 1000)$ to $(3000, 3000)$. FluidiCL outperforms CPU-only and GPU-only performances by a huge margin, with the geomean speedup being $1.4\times$ over the best of the two devices.

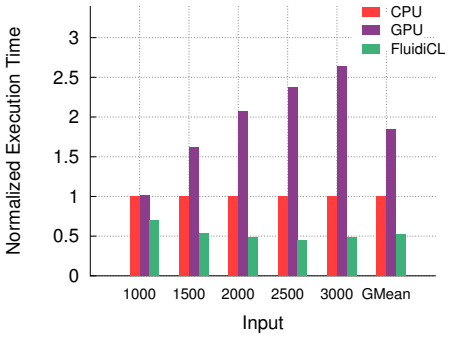

Figure 14: Performance of SYRK on different inputs. Total running time normalized to the best among CPU-only or GPU-only in each case is shown.

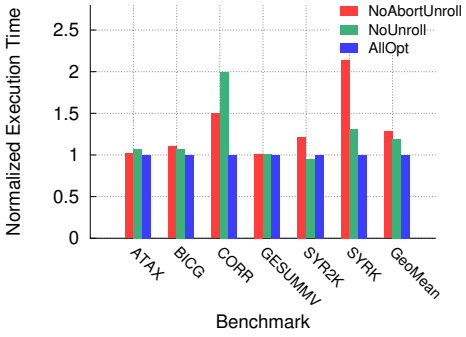

Figure 15: Effect of Work-group abort inside loops and loop unrolling. Total Running Time normalized to the run with the work-group abort optimization is shown.

9.3 Effect of Optimizations

The performance described in Figure 13 has all but one optimizations enabled. Now we show the individual effect these optimizations have on the performance.

We show the importance of work-group aborts inside loops in GPU kernels. The *NoAbortUnroll* bar in Figure 15 shows the effect of having work-group abort checks only at the beginning of a work-group. Almost all benchmarks show improvement in performance when the optimization is enabled (the *AllOpt* bar), with CORR, SYRK, and SYR2K showing significant improvement. This is because letting loops run to completion even when the CPU has already finished executing the work-group leads to wasted work and the opportunity to let the GPU kernel terminate early is lost. The figure also demonstrates the effect of including work-group aborts inside loops but not doing loop unrolling (the *NoUnroll* bar). Five out of six benchmarks would experience slowdown because the abort check introduced inside loops no longer allows the GPU compiler to perform loop unrolling.

Finally, to demonstrate that FluidiCL has the capability to automatically choose the best performing kernel among different versions, we provide the benchmark CORR with an improved (hand-

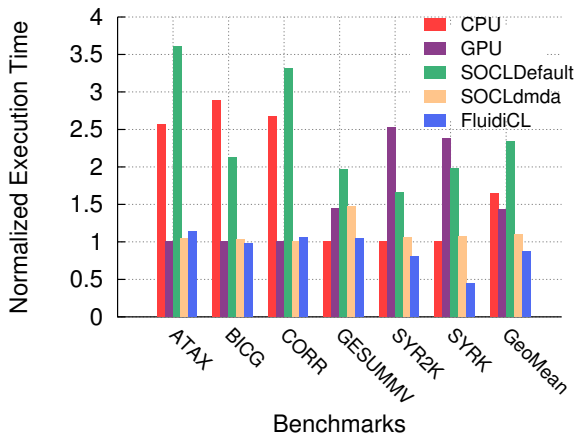

Figure 16: Comparison with SOCL. Total running time normalized to the best among CPU-only or GPU-only in each case is shown.

	GPU	CPU	FluidiCL	FCL+Pro
Time (s)	7.04	18.78	7.42	2.53

Table 3: Performance of CORR when provided with a choice of kernels. (FCL+Pro) shows the performance with online profiling in FluidiCL.

written) kernel for the CPU that performs significantly better than the baseline and show the ability of FluidiCL to identify it using online profiling. This alternative kernel has its loops interchanged for cache locality. As seen in Table 3, FluidiCL is able to use the best performing kernel and perform $2.9\times$ better than with the baseline kernel. This demonstrates that FluidiCL is able to pick the best running CPU kernel automatically when presented with a choice of more than one.

9.4 Comparison with StarPU

StarPU [1] is a heterogeneous runtime system that provides task programming API which the developers can use to define tasks and the dependencies between them. These tasks are scheduled by the StarPU runtime on available heterogeneous devices. In this section we compare the performance of FluidiCL with SOCL – the OpenCL extension to StarPU. SOCL eliminates the need for writing StarPU API by providing a unified OpenCL runtime which in turn invokes the necessary StarPU API for scheduling and data management. Since unmodified OpenCL applications can use SOCL to run OpenCL programs on CPU and GPU, it is similar in nature to what FluidiCL attempts to do.

In Figure 16, we compare the performance of SOCL using the default eager scheduler of StarPU, referred to as SOCLDefault and the more advanced dmda scheduler. The results shown are execution times normalized to the best among CPU–only and GPU–only. We see that FluidiCL significantly outperforms the eager scheduler of StarPU in every benchmark. For benchmark SYRK, FluidiCL is faster than SOCL by more than $4\times$.

The dmda scheduler requires a performance model for each application to be built by running a calibration step. This calibration step involves running the application with at least ten different input sizes. Later, when each application is run with calibration turned off, SOCLdmda uses the performance model from the calibration runs and schedules work using the model. We ran the benchmarks with SOCLdmda calibrated with at least 10 initial runs and a final run where the performance is measured. As seen in Figure 16, FluidiCL outperforms SOCL-dmda also in most benchmarks with the benchmark SYRK being more than $2.4\times$ faster. In case of ATAX and CORR, FluidiCL runs within 9% of SOCL. The carefully implemented FluidiCL runtime is able to dynamically assess the execution characteristics of a program on each device and manages to consistently outperform the more general heterogeneous schedulers in StarPU indicating its effectiveness in the CPU–GPU environment. Note that FluidiCL is able to match or exceed the performance of the StarPU scheduler without the need for prior profiling or calibration. We believe this feature of FluidiCL is important especially from the point of view of being able to run applications and get good performance without having to tune/calibrate the runtime for each application.

9.5 Sensitivity to Chunk Size and Step Size

The above results were obtained with the initial *chunk size* set to 2% of the work-groups for the CPU subkernel and progressively increasing the allocation by a *step size* of 2%. For example, if the initial work-group chunk size is 2% and the step size is 2%, the first CPU subkernel is launched with 2% of the work-groups, and the subsequent subkernels with 4%, 6% and so on. First, we demonstrate the results of FluidiCL for all the benchmarks when the initial chunk size is varied with different values between 1% and 75%, with a step size fixed at 2% (Figure 17). We report the execution time in each case normalized to the execution time with both chunk size and step size set to 2%. We find that larger initial chunk sizes (greater than 10%) perform poorly in case of BICG, SYR2K and SYRK because these are benchmarks where both the CPU and the GPU executing together gives the best performance, and having too large a chunk size means that the results computed by the CPU are not communicated with the GPU regularly enough. However in case of GESUMMV, larger initial chunk sizes perform better. This is because the benchmark runs best on CPU alone and having a higher chunk sizes reduces the overheads associated with launching more CPU subkernels. Even in this case, we find that the value chosen in our experiments (chunk size of 2%) performs within 1% of the best performing chunk size.

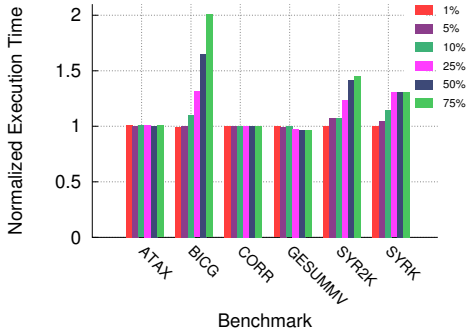

Figure 17: Sensitivity to initial chunk size. Total running time normalized to a chunk size of 2% is shown.

To determine the sensitivity of FluidiCL to the step size chosen, we run FluidiCL with an initial chunk size of 2% and different step sizes between 0% and 9%. A step size of 0% means that every CPU subkernel is launched with 2% work-groups and this allocation does not change. Figure 18 shows the results normalized to the step size of 2% chosen in all our experiments. We find that this value comes to within 1% in most cases with the maximum degradation being 3%.

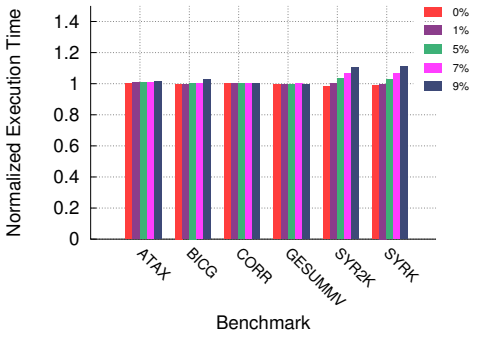

Figure 18: Sensitivity to step size. Total running time normalized to a step size of 2% is shown.

10. RELATED WORK

There have been examples of manual implementations of CPU–GPU work distribution like the one by Intel [9] which shows the benefit of using these devices simultaneously. A runtime like FluidiCL can greatly simplify the effort in such cases.

Qilin [20] is a framework for CPU–GPU systems which relies on offline profiling to build an analytical model which is used to perform adaptive mapping for programs at runtime. Programs must be written to use Qilin API and data structures in order to use this framework whereas FluidiCL works on existing OpenCL programs without any modification. Ravi et al. [23] present a dynamic work-distribution scheme for a class of applications called generalized reduction. Unlike this work, FluidiCL is more general and applies to any program written in OpenCL, since a simple work-sharing scheme they describe is unlikely to benefit in a more general setting. Grewe et al. [7] start with an OpenMP program and use a machine-learning based prediction to determine whether to run the program on the CPU or run a translated OpenCL version on the GPU. This work does not use both the CPU and the GPU together to accelerate a kernel like FluidiCL does. Grewe et al. [6] describe a static task partitioning scheme for OpenCL programs on CPU–GPU systems. The work uses static code features of programs to train a machine-learning based model, which is then used to determine the best partitioning of a different program. However unlike FluidiCL, their work only considers programs with single kernels and therefore cannot handle the complications that arise due to data management in programs with multiple kernels. Also, it is not clear whether their approach supports appropriate data merging scheme for coherence management. The training required by the machine-learning model makes it less portable than our scheme.

New programming models, API extensions and programmer annotations have been proposed to support multiple homogeneous or heterogeneous devices. StarPU [1], HDSS [2] and XKaapi for multiple GPUs [4] are some examples. Compared to these, FluidiCL requires *no change* to existing OpenCL programs. The SOCL extension [25] to StarPU attempts to provide similar functionality to StarPU for OpenCL programs. We have compared the performance of FluidiCL and SOCL in Section 9.4. Other proposals include extensions to accelerated OpenMP [24] for heterogeneous execution on CPU–GPU systems and OmpSs [3] for GPU clusters, which require the programmer to use OpenMP-like clauses for mapping a parallel region/function to the GPU and indicate the data which is read/written by it.

Zhang et al. [27] demonstrate the use of CPU for removing dynamic irregularities in GPU programs. However, the work does not run the application on both devices. Hardware schemes that utilize CPU for prefetching GPU data have also been proposed [26].

Kim et al. [13] propose a framework for OpenCL programs that uses multiple GPUs but presents a single device image to the user. It performs buffer range analysis to determine the partition that results in the least data transfer overheads. This work uses multiple identical GPUs and hence does not encounter the problem of devices with different capabilities. Also, some of these buffer range analysis techniques are orthogonal to our work and could be integrated into FluidiCL to reduce the amount of data transferred between devices. SnuCL [14] presents an OpenCL framework for heterogeneous clusters. It makes CPUs and GPUs present on a different node in a cluster appear as local devices for OpenCL programs. However the task of mapping computation and data to these devices is left to the programmer, whereas in case of FluidiCL, this is exactly the problem we are trying to tackle in a single node CPU–GPU environment. SKMD [18] proposes an OpenCL runtime for heterogeneous devices which, like FluidiCL, takes a kernel written for a single device and executes it across multiple devices and takes care of data merging automatically. However, unlike FluidiCL, it requires prior profiling of the programs in order to make the scheduling decision across devices. Other efforts in implementing OpenCL runtimes include an OpenCL runtime for Intel's Single-chip Cloud Computer (SCC) [16], for CellBE processors [17] and for improving the performance of GPU kernels on CPUs [8]. FluidiCL can be extended to devices like Cell BE and SCC, as well as to support multiple heterogeneous devices.

11. CONCLUSION

In this paper, we have presented FluidiCL – a heterogeneous OpenCL runtime that targets CPU–GPU systems. FluidiCL takes applications written for a single device and automatically runs each kernel on both the CPU and the GPU in a coordinated fashion and handles data management in a transparent manner. While performing work-distribution, FluidiCL intelligently factors in data transfer overheads so that the performance of the application always does nearly as well as the faster of the two devices. In applications where simultaneous CPU–GPU execution helps, FluidiCL achieves a performance improvement of upto $2.24\times$ over the best of the two devices. We have shown that the FluidiCL adapts very well to changes in input size as well. We have also implemented several optimizations in our runtime to improve overall performance and shown the effects these optimizations have on the performance of the benchmarks. With optimizations in place, we have shown that FluidiCL gets a geomean speedup of nearly 64% over a high-end GPU and 88% over using a quad-core CPU for all applications. When compared with the best of the two in each case, FluidiCL still gets a geomean speedup of 14%. We have also compared FluidiCL with other similar approaches (SOCL) and demonstrated that FluidiCL outperforms them without requiring any tuning/calibration steps of SOCL.

Future work involves incorporating compiler or runtime techniques to reduce data transfers done between the CPU and the GPU. We plan to use a source-to-source compiler to fully automate the code transformations and optimizations applied in this work. We also plan to add support for multiple heterogeneous devices in future.

12. ACKNOWLEDGMENTS

We are grateful to AMD India for partially funding this work. We thank the anonymous reviewers for their suggestions and comments. We also thank Sreepathi Pai and other HPC Lab members

for the technical discussions and help in improving the quality of the work.

13. REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, Jan. 2013.

[3] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, 2012.

[4] J. V. Ferreira Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *24rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 75–82, Columbia University, New York, États-Unis, Oct. 2012.

[5] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[6] D. Grewe and M. F. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *CC '11: Proceedings of the 20th International Conference on Compiler Construction*. Springer, 2011.

[7] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO '13: Proceedings of the 11th International Symposium on Code Generation and Optimization*. ACM, 2013.

[8] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[9] Intel. Intel Launches SDK for OpenCL Applications at SIGGRAPH 2012, 2012.

[10] Khronos Group. Conformant Products, 2012.

[11] Khronos Group. OpenCL 1.2 Specification, 2012.

[12] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems, 2013.

[13] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[14] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[15] L. Nyland, M. Harris, J. Prins. Fast N-Body Simulation with CUDA, 2012.

[16] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee. An opencl framework for homogeneous manycores with no hardware cache coherence. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 56–67, 2011.

[17] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.

[18] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[19] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[20] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[21] NVidia. NEW TOP500 LIST: 4X MORE GPU SUPERCOMPUTERS, 2012.

[22] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[23] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM.

[24] T. Scogland, B. Rountree, W. chun Feng, and B. De Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144–155, 2012.

[25] Sylvain Henry. SOCL – OpenCL Frontend for StarPU, 2013.

[26] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[27] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.