Dynamic Task Parallelism with a GPU Work-Stealing Runtime System

Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar

Department of Computer Science, Rice University
{cs20,jmg3,alina,vsarkar}@rice.edu

Abstract. NVIDIA's Compute Unified Device Architecture (CUDA) and its attached C/C++ based API went a long way towards making GPUs more accessible to mainstream programming. So far, the use of GPUs for high performance computing has been primarily restricted to data parallel applications, and with good reason. The high number of computational cores and high memory bandwidth supported by the device makes it an ideal candidate for such applications. However, its potential for executing applications that may combine dynamic task parallelism with data parallelism has not yet been explored in detail, in part because CUDA does not provide a viable interface for creating dynamic tasks and handling load balancing issues. Any support for dynamic task parallelism has to be orchestrated entirely by the CUDA programmer today.

In this work we extend CUDA by implementing a work stealing runtime on the GPU. We introduce a *finish-async* style API to GPU device programming with the aim of executing irregular applications efficiently across multiple shared multiprocessors (SM) in a GPU device without sacrificing the performance of regular data-parallel applications within an SM. We present the design of our new intra-device inter-SM work-stealing runtime system and compare it to past work on GPU runtimes, as well as performance evaluations comparing execution using our runtime to direct execution on the device. Finally, we show how this runtime can be targeted by extensions to the high-level CnC-CUDA programming model introduced in past work.

1 Introduction

Graphics Processing Units (GPUs) contain hundreds of lightweight cores with large bandwidth access to on-chip memory. The massive parallelism and memory bandwidth of a GPU makes it very useful for computationally heavy applications operating on large data sets. Add to this the relative energy efficiency and low cost of its hundreds of simple cores compared to a CPU, with fewer cores, and we begin to understand its growing applicability both on the supercomputer scale [1] and in desktop computing for application areas that include life sciences, medical imaging and finance [2].

NVIDIA provides a C/C++ based API for programming their GPUs called the Compute Unified Device Architecture (CUDA) programming model. CUDA

includes explicit memory management functions and generally requires considerable knowledge and understanding of the GPU hardware to achieve the significant performance gains that GPUs are capable of delivering, representing a steep learning curve to many programmers. The CUDA programming model was developed to primarily benefit regular data parallel applications, that are aligned to the needs of heavy graphics processing. In a nutshell, CUDA allows the programmer to launch large batches of SIMD threads. These collections of threads are decomposed into blocks of threads, containing anywhere from 32 to 1024 threads. The threads within a block are all mapped to the same streaming multiprocessor on the device. Threads on the same streaming multiprocessor (SM), and therefore in the same block, execute the same kernel in lock-step, but threads in different SMs may run completely separate kernels with no performance penalty.

Executing irregular applications that may involve dynamic task parallelism is not trivial using the current CUDA API. With this work, we aim to help improve the adoption of CUDA-like models for irregular applications by handling many of the lower level runtime details for the programmer. We have designed and implemented a runtime abstraction for dynamic task parallelism using the *finish-async* style API [3] on top of the regular data parallel model of execution. The programmer is freed from the responsibility of load balancing dynamically created tasks with the aid of our CUDA work stealing scheduler that operates across multiple SMs in the same device. The runtime helps reduce data transfer overhead by overlapping it with kernel execution, manage multiple devices, and distribute tasks on the device with the goal of balancing the workload across all SMs on a GPU.

The rest of the paper is organized as follows. Section 2 presents the design of our new GPU runtime for dynamic task parallelism. Section 3 provides implementation details of the work-stealing runtime deployed using the CUDA programming model. Section 4 shows how the runtime can be targeted by extensions to the high-level CnC-CUDA programming model introduced in past work [4]. Section 5 presents the results of experiments on our runtime system. Finally, Section 6 discusses related work and Section 7 contains our conclusions.

2 Design of GPU Work-Stealing Runtime System

GPUs' restriction to primarily data parallel applications means its potential for executing irregular applications that combine dynamic task parallelism with data parallelism has not yet been explored in detail. With CUDA currently providing no viable interface for dynamically creating tasks or handling load balancing issues, it may be some time before any official support is provided, if ever. In applications such as quicksort (say), each step may produce tasks that execute in parallel. This requirement for dynamic task creation is not trivially solved using the current CUDA API. Another factor that inhibits execution of irregular applications on the GPU is the need for task synchronization. CUDA allows synchronization only among threads that belong to the same block, which

can run on only one SM. As a result, the only way that parallel blocks of threads can synchronize with each other in CUDA is via multiple kernel launches. This enforces a severe restriction on running irregular applications, that may require combining results from parallel work in each step before moving onto the next step.

Both of these disadvantages, dynamic task creation and parallel task synchronization, can be addressed by the *finish-async* style of programming [3]. In this model, an *async* creates or spawns a task that can potentially execute in parallel with the continuation of the spawning task. The *finish* provides a scope for all spawned tasks, both direct and nested, to complete before execution of the continuation of a finish. The *finish* and *async* constructs provide a natural way for programmers to express task parallelism using dynamic task creation and synchronization.

In this work, we have developed a runtime system on the GPU that provides a task-based implementation for abstractions such as finish and async. The goal of our *finish-async* model on the GPU is to be faster than the current divergent execution model for irregular applications without sacrificing the performance of intra-SM regular computations. We aim to provide the users with a simpler programming model for task parallelism, while handling the problem of load balancing which all systems supporting dynamic task creation must deal with. At the moment, the finish-async functionality on the device has only been thoroughly tested with a flat finish wrapping all device asyncs.

3 Implementation Details of GPU Runtime

The goal of this runtime is to balance work across the threads of a GPU better and with less effort for the user than a hand written application, while supplying a simpler and easier to use API. To achieve this goal we use a hybrid task distribution model that uses both work stealing and work sharing queues to provide load balancing between SMs on a CUDA device, and across different devices.

Our runtime starts by launching N blocks of CUDA threads on each device, where N is the number of SMs on that device. Conceptually, our runtime treats these blocks as worker blocks, analogous to worker threads in CPU-based work-stealing runtimes. Each of these worker blocks executes the runtime kernel. Shown in Figure 1, each worker block maintains its own work stealing deque. A worker can also steal from other workers' deques that reside on the same device. A separate FIFO shared queue is maintained to place tasks from the host onto the device. Only the host can push tasks onto this queue while the workers on the device compete to pop these tasks. At the moment, tasks are distributed evenly and naively among devices from the host, but more intelligent device selection for task placement would be an interesting direction for future work.

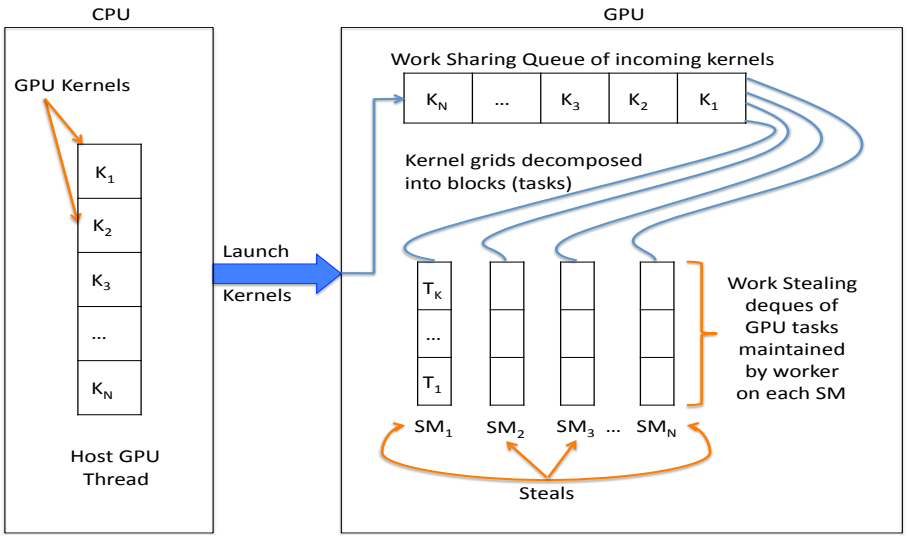

Fig. 1. GPU runtime for dynamic task parallelism

3.1 Task Representation

As mentioned earlier, CUDA has two levels of parallelism: SIMD threads within a block of threads on the same SM, and threads executing potentially different kernels on different SMs. When we talk about tasks in this paper, these represent tasks which are run by a block of threads, not individual threads, though tasks can be created by any thread in a block. To dynamically create tasks at the fine grain level of individual threads would be prohibitively expensive for GPUs, in most cases leading to major divergence and serialization of execution.

Tasks designated for execution on the device are represented by a task structure. This task structure uses a param structure to represent inputs and outputs of the device task. The work sharing and stealing deques contain pointers to these task structures, shown below.

```
typedef struct {
        void *ptr; // address of the data
        size_t size; // number of bytes in this parameter
        volatile int *done_flag; // Indicates readiness of data
        unsigned char type; // Indicates input and/or output from device
} param;

typedef struct task_ {
        int type; // what code to run for this task
        param *p; // list of parameters to this task
        int *ready_flag; // indicates if this task has completed
        int num_params; // number of parameters
        ...
} task;
```

The task structure represents a task on the device with certain global inputs and outputs. The type field of a task structure uses an integer id to identify the code to be executed by this task, while the param pointer points to a list of parameters containing information on the inputs and outputs to the device. Tagging a parameter as being input or output only has significance in context of the device, and does not describe its relation to a task. The ready_flag field is used to test when a continuation task is ready to be executed in our async-finish model for on-device dynamic task creation, which we will discuss in more depth later in this section.

3.2 Task Creation

A task is launched by building a task structure with the appropriate parameters and type on the host or on the device and then placing it on the work sharing queue (if launching from the host) or on the worker block's work stealing queue (if launching from the device). If launched from the host, this process includes asynchronously allocating and copying the input and output data.

Once this task is placed in the appropriate queue it will then be fetched by a worker block. The worker block will then call the appropriate function based on the type field of the task. All queues on the device use locks to protect against concurrent conflicting accesses using CUDA's atomic CAS instruction.

A large obstacle to launching tasks from the device lies in CUDA's lack of dynamic memory allocation. Without dynamic memory allocation and with a potentially dynamic number of tasks it is impossible to estimate the amount of device memory that will be necessary to preallocate for each application. At the moment, this problem is being solved by a mock memory manager for the device, implemented as a linked list of preallocated task structures for each type of task on each device. Each of these empty tasks is allocated with memory for its parameters. When a device on the task launches a nested task it simply allocates a task from one of these linked lists, which can then be pushed onto the work stealing queues in the same manner as any task. When a task has completed on the device, it can be freed for future use by placing it back onto these linked lists.

Once all tasks have been placed on the device, the worker blocks are told that there is no remaining work by placing a special value into the work sharing queue. Upon finding this value, each worker block can be sure that there are no incoming tasks from the host to be run and instead begins waiting for a global counter of tasks to reach zero, while continuing to attempt to pop from its own work queue and steal from others' work queues.

3.3 Communication Between Device and Host

One of the largest burdens placed on a CUDA programmer trying to achieve optimal execution on the device is device memory management. While the most

basic memory management functions are easy to work with (cudaMalloc and cudaMemcpy being analogous to malloc and memcpy on the host), they also generally result in inefficient executions with lots of blocking function calls. Therefore, managing device memory for the user is a problem that any CUDA runtime system must solve. Ours hides all device allocation and communication from the user, instead using the contents of each task structure to know what the memory requirements of a task are.

One of the keys to good performance in any multicore system is overlapping communication with computation in order to hide the added overhead that isn't a concern in sequential code. In this runtime system, we leverage CUDA streams in an attempt to manage this overlapping for the user. Each time a new task is placed onto the device from the host, the associated input is copied with it. Using asynchronous memory copies in a CUDA stream allows this communication to occur in parallel with the runtime kernel and any programmer-written code executing on the device. In the future, a more advanced task pushing mechanism on the host could also allow copying through multiple streams at once to ensure maximum utilization of the bandwidth to each device.

In this implementation, a list of address mappings conceptually sits between the host and device memory. These mappings allow the runtime on the host to keep track of what host memory locations have already been copied to the device, what device location they were copied to, and how many bytes were transferred. This information allows us to be certain of where to copy to/from and how much to copy.

3.4 PTX Usage

During the implementation, we made considerable effort to minimize the overhead our runtime would incur on the computational resources of the device. One technique we used was to implement the entire work-stealing/work-sharing runtime kernel in NVIDIA's Parallel Thread Execution (PTX) language, a lower level language alternative to the C API that is normally used in CUDA. There are several advantages to using PTX. First, it provides more direct control over the load instructions being used to ensure that no stale values were being read from shared locations in global memory (i.e. from the work-stealing and work-sharing queues), without having to use atomic instructions. For instance, extensive use was made of ld.cv (a cache volatile load) in the runtime kernel. Second, we were also able to manually limit the number of registers being used. Minimizing register usage is often critical for good performance in CUDA. Each SM has a fixed number of registers which are distributed among the threads mapped to it, so by decreasing the number of registers a thread uses when executing a kernel it becomes possible to run more threads at once on a SM. Third, using PTX permitted the manual optimization of the computation at the instruction level.

While this was a design decision made at the time, it would be possible to reimplement the same runtime using CUDA C either with inline PTX or only atomic instructions. We are also interested in the possibility of creating an OpenCL compatible implementation, as future work.

4 Programming model

The GPU work-stealing runtime is a standalone tool which can be integrated with a programming model in order to provide a friendly user-interface. We made the integration with a new C-based implementation of the Concurrent Collections (CnC) programming model, being motivated by the results in previous work on CnC-CUDA [4]. The details of CnC-C are beyond the scope of this paper. Current work is being done on the integration with the Habanero C language [5] which uses the async-finish model, making integration more straight forward. On the other hand the CnC model is more general than the async-finish model; that is to say more graphs/programs can be expressed using CnC than with finish-async. In our work we will be using a subset of CnC's synchronization pattern. We reserve for future work extending the current implementation to support data dependences between CPU and GPU tasks.

CnC offers a easy way for the programmer to specify the dependences within his program with the aid of an intuitive graph language. The main components of any graph are data collections, control collections and steps. Data collections can be viewed as a tagged data storage (analogous to key-value pairs). A tag's role once it is put into a control collection is that of starting (prescribing) the steps assigned to it. Steps represent units of computation and are prescribed by a tag. They also read (get) items from data collections and can put items and/or tags into their data and control collections. A step may request an item with a certain tag from a data collection without having the knowledge whether the item exists or not. The CnC runtime will ensure the steps that have been prescribed will execute when the data they need is available.

Let us take one of the benchmarks - Crypt - and show the transformation of a CnC graph to its analogous CUDA code, assuming we already have a kernel written in CUDA for Crypt. First, we will write a simple CnC graph. The notation '::' in a CnC graph indicates the prescription of a computation step with a tag. The notation '− >' in a CnC graph indicates a computation step which either consumes or produces an item. For example, in the CnC graph below representing the crypt application, decrypt_tag is a tag collection which prescribes (launches) the computation step gpu_decrypt. The data collections used are "original"the original text, "z"the encryption key, "crypt"the encrypted text, "dk"the decryption key and "original_decrypted"the original text after decryption.The gpu_encrypt computation step consumes items from data collections original and z, and produces items into crypt and decrypt_tag output collection, where decrypt_tag is a control collection.

```
<decrypt_tag>::gpu_decrypt;
<encrypt_tag>::gpu_encrypt;
[ original ], [ z ] -> { gpu_encrypt } -> [ crypt ], < decrypt_tag >;
[ crypt ], [ dk ] -> { gpu_decrypt } -> [ original_decrypted ];
```

Using features offered by CnC much of the code needed to link the user's inputs with the kernel will be auto-generated.

A CnC program would then be written to work with the code generated by this graph file. The C code will look as follows:

```
CnCGraph graph;
graph.original.Put(tag, orig);
graph.z.Put(tag, z);
graph.encrypt_tag.Put(tag);
```

The CnC-C runtime will then manage the data dependencies, control dependences, and computation step invocation using the Habanero-C parallel programming language. Integrating the GPU work-stealing runtime with the CnC-C programming model would allow computation analogous to the below CUDA code to be generated for the user from the CnC graph specified above:

```
cudaMalloc(&d_original); cudaMalloc(&d_crypt);
cudaMalloc(&d_original_decrypted);
cudaMalloc(&d_z); cudaMalloc(&d_dk);
cudaMemcpy(d_original, original);
cudaMemcpy(d_z, z); cudaMemcpy(d_dk, dk);
encrypt_kernel<<<blocks_per_grid,threads_per_block,0,stream>>>(d_original,
    d_z, d_crypt);
decrypt_kernel<<<blocks_per_grid,threads_per_block,0,stream>>>(d_crypt,
    d_dk, d_original_decrypted);
cudaMemcpy(original_decrypted, d_original_decrypted);
```

The eventual goal is to auto-generate all code related to the CUDA runtime for the CnC user, only requiring a) an initialization call in the CnC Main function, b) a terminating call to signal the runtime kernel on the device to exit, and c) CUDA kernels to be used as computation steps. Reaching complete auto-generation is still a work in progress, but a manual proof of concept has already successfully demonstrated the integration of the GPU runtime and CnC.

5 Experiments

To test the performance of our GPU work stealing runtime, we tried to find examples of applications that are challenging to implement efficiently on graphics hardware and data parallel applications that are already well suited for CUDA. We use *n-queens* from BOTS [6] for our first benchmark. The BOTS implementation of nqueens on a CUDA device is difficult because it can result in unbalanced computation trees and requires a lot of dynamic task creation and load balancing. For our second benchmark, we will use an implementation of the *quicksort* sorting algorithm based on [7]. Our third benchmark, the *crypt* benchmark from the Java Grande Forum Benchmark Suite [8], is regular and recognized to be a good candidate for GPU execution. Finally, our fourth benchmark will be a shortest path computation based on the implementation of *Dijkstra's* algorithm in [9]. This benchmark starts with a single task and must then spread the load across all SMs as evenly as possible.

In our tests, we compare runs with different numbers of devices as well as different data sizes to see how this impacts execution time. Additionally, we use diagnostic data from our runtime to measure how effectively we are load balancing the application's work.

Benchmark tests were performed with 1, 2, or 3 NVIDIA Tesla C2050 GPUs. The Tesla GPUs were tested with direct calls to runtime functions from C or CUDA code. Each Tesla C2050 has 14 multiprocessors, 2.8 GB of global memory, 1.15 GHz clock cycle and are using CUDA Driver and Runtime version 3.20. The host machine of the Tesla GPUs has 4 Quad Core AMD CPUs (2.5 GHz).

5.1 N-Queens

For the n-queens benchmark we ported the BOTS implementation of n-queens to the device using our runtime. The BOTS benchmark suite is intentionally designed to test the effect that irregular parallelism has on a multicore system. Irregular and recursive based parallelism results in less predictable numbers and distributions of tasks that are more difficult to allocate between worker threads. Because of this, n-queens is a challenge to port to the CUDA programming model without a loss in performance. Our runtime facilitates this irregular parallelism on graphics hardware. In Figure 2 we can see that our implementation of the n-queens benchmark scales well across multiple devices. From experience in developing it, we can also say that building an n-queens benchmark for the CUDA was much simpler with our runtime than it would have been without.

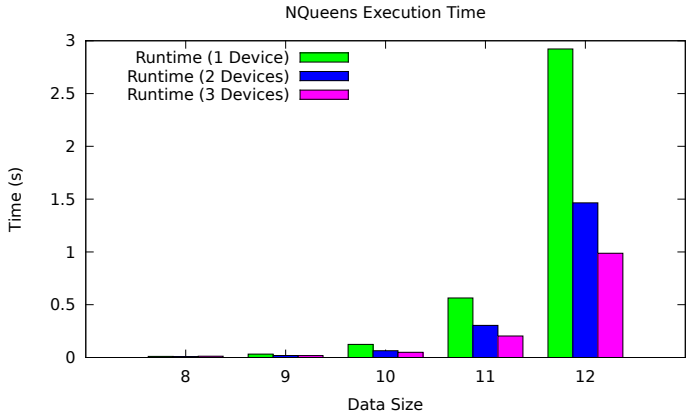

Fig. 2. Execution times in seconds of the n-queens benchmark using our work stealing GPU runtime on 1, 2, or 3 devices.

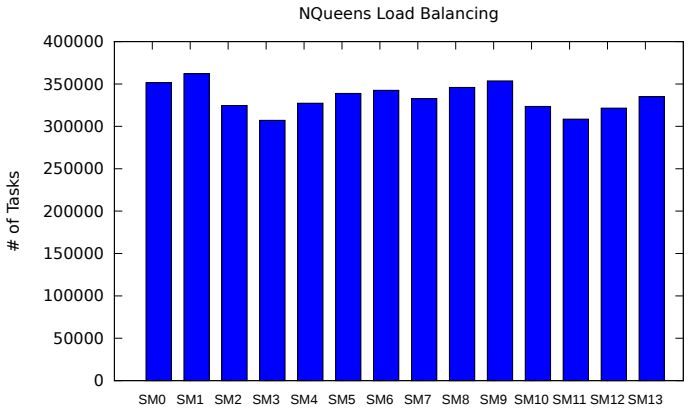

Fig. 3. Tasks executed by each SM on a single device running the n-queens benchmark.

5.2 Quicksort

Our quicksort implementation is based on the algorithm described in GPU-Quicksort [7]. We use it as another demonstration of our runtime's ability to handle irregular parallelism on a device intended for extremely data parallel applications. Note that the Cederman results listed in this table are from an older model graphics card. However, the code used in Cederman is a *highly* optimized quicksort implementation which we are able to demonstrate a performance advantage over with our more naive implementation at larger data sets. Any performance advantage that the Cederman results have are a function of the time spent on the different implementations and any overhead of our runtime. Future work would include requesting a copy of the original source code and testing our runtime with it.

5.3 Crypt

The crypt benchmark from the Java Grande Forum Benchmark Suite (JGF) performs the IDEA cryptographic algorithm on a block of bytes, encrypting and then decrypting and validating the results. This algorithm is already well suited for execution in a data parallel programming model like CUDA. The encryption and decryption of every 8 bytes can be run independent of the rest of the data set with no conflicting accesses to shared variables. We include crypt in these experiments to demonstrate that using this runtime to run an application which is already well suited for CUDA will not result in significant degradation of performance.

The hand coded CUDA version of crypt which we tested against does not try to take advantage of any overlapping of communication and computation. The reason for this is that in our initial implementation we did not predict any

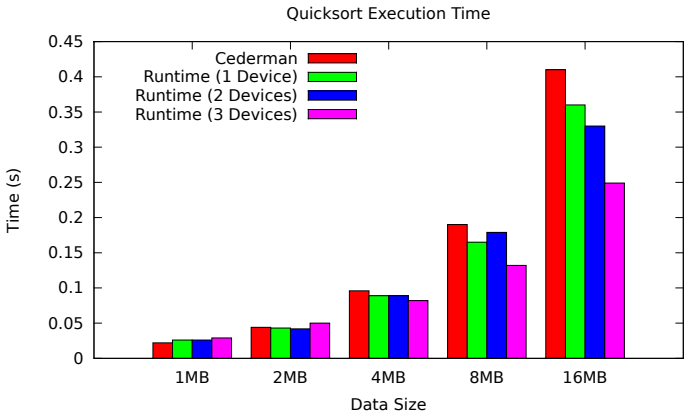

Fig. 4. Execution times in seconds of the Quicksort benchmark using our work stealing GPU runtime on 1,2, or 3 devices compared to execution times in Cederman and Tsigas.

advantages in splitting the copying of the original data to the device. Later tests showed that better performance was achievable with hand coded CUDA code, but required considerable more experience and effort on the part of the CUDA programmer.

Though not shown in Figure 5 an unexpected benefit of our runtime automatically providing device management for more than one device was the ability to handle larger data sets when using our runtime.

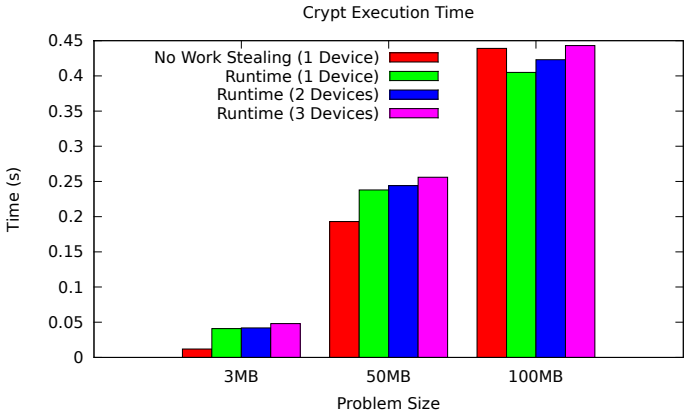

Fig. 5. Execution times in seconds of Crypt benchmark using our work stealing GPU runtime on 1, 2, or 3 devices and using hand coded CUDA on a single device.

5.4 Dijkstra's Shortest Path Algorithm

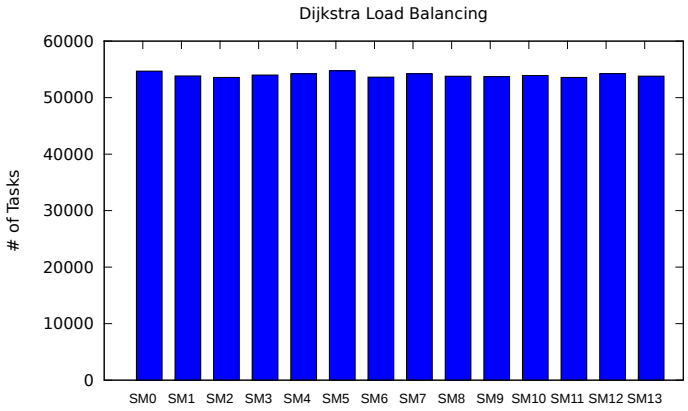

Fig. 6. Tasks executed by each SM on a single device running the Dijkstra benchmark.

We implemented Dijkstra's shortest path algorithm using our runtime based on the algorithm used in "Dynamic Work Scheduling for GPU Systems" [9] for the same reason as they did: it is an application which effectively tests the load balancing abilities of a runtime. Figure 6 shows how many tasks each worker executes while finding the distance from each node to a destination node in a 10,000 node bidirectional weighted graph. Initially a single task is placed on a single worker. From there, our runtime is able to distribute tasks to all SMs on the device, indicated by the level top surface of Figure 6.

5.5 Multi-GPU Performance

For many of the above benchmarks, we see less than expected acceleration from using multiple devices, a counterintuitive result which requires some explanation. The primary cause of this is redundant memory copies. Take the crypt benchmark for example. In crypt, there are encryption and decryption keys being used which must be accessible from each device. This means that by increasing the number of devices, you are actually increasing the amount of necessary communication that is necessary. Other factors may be causing this discrepancy as well, but further investigation would be necessary to understand what they may be. Two potential solutions to this problem could be: 1) more intelligent task placement of tasks which share data on the same device, or 2) use CUDA 4.0's unified virtual device address space to access a single copy from any device. The usefulness of more intelligent task placement when benchmarking would be minimal as it would mean even when 3 devices are initialized not all would be necessarily

used. Additionally, while early experience with the unified virtual address space shows that it is more useful in the easing porting, its use can actually result in performance degradation.

6 Related work

There have been some recent experiments at either including GPU execution into current programming models and languages, or implementing a task-parallel runtime on the GPU. Lastras-Montano et al. [9] implemented a work stealing runtime on the device. In their paper, a worker is a single warp of threads (32 threads). Each of these warps is assigned a q-structure, which is a collection of queues in shared and global memory to place tasks in and steal tasks from. Since multiple warps in a block always reside on the same SM, this runtime can attempt to steal from those other warps via faster on-chip shared memory before looking to steal from global memory (i.e. from workers in other SMs). Their use of queues in shared memory would yield considerably less latency than our global memory queues, and could be included in future work. However, designating each individual warp as a worker would lead to increased contention for steals. While our runtime is designed for continuous use throughout an application, their runtime starts with a kernel launch and ends when a certain number of steals have failed. This termination condition could be harmful to performance critical applications.

The X10 programming language recently began supporting the execution of tasks on CUDA devices [10]. They do not provide an actual on-GPU work stealing runtime, but instead integrate GPU tasks into their host work stealing runtime. They provide the user with a simpler API for allocating device memory and copying asynchronously from host memory to device memory than the CUDA API does and expose the block and thread CUDA memory model to the programmer in what might be a more intuitive way: as nested loops iterating over X10 points. However, this is not a device runtime and even though the appearance of the code may be more familiar to non-CUDA programmers and they hide some of the memory management from the programmer, the programmer must still be very aware of the CUDA programming model and its challenges and nuances.

The work in [11] presented a variety of potential GPU load balancing schemes ranging in complexity from a static array of tasks to a work-stealing task distribution technique similar to the one used in our runtime system. In order to compare the relative performance of the different systems proposed in their paper they used an octree creation application. They demonstrated that the task distribution system most similar to our own methods for distributing tasks between SMs on the same device achieved the best performance of those tested.

StarPU [12] [13] is another runtime system for hybrid CPU and GPU execution. Similar to X10, this system's role is to dispatch tasks to different processing unit for which it makes complex scheduling decisions based on different hardware. StarSs and its GPU extension GPUSs [14] is building on the OpenMP

model and offers simplicity by using pragmas to define tasks that can be executed on the GPU. However each task annotated for GPU execution will run as an independent kernel without any control on how the computation is distributed on the device. Both StarPU and StarSs will use the CPU, GPU as well as other resources like the Cell to achieve system-wide load balancing, but neither of them addresses the problem of load balancing inside the GPU, but base their assumption on the fact that work inside the GPU kernel will be uniform.

This work has the potential of being supported on OpenCL [15].In such a scenario, slight modifications of the runtime API will needed to be done for conformance to OpenCL standards, and possibly a reimplementation of the runtime kernel.

7 Conclusions and Future Work

In this paper, we have presented a GPU work stealing runtime that support dynamic task parallelism at thread block granularity. We demonstrated the effectiveness of our combination of work stealing and work sharing queues in distributing tasks across the device using the nqueens and Dijkstra benchmarks. Each of these benchmarks starts with a single task on a single SM, requiring low overhead task distribution to achieve good performance. We demonstrated that even applications for which CUDA is well suited using this runtime incurs little overhead and may even result in better performance, a result of automatically managing multiple devices for the user as well as overlapping data transfer with kernel execution. We gave a brief overview of other simplified interfaces to the device that are currently available or in development and compared them to our own approach. While support for CUDA calls in X10 provide simpler access to the device and the previous work by Angels et al provided fine grain load balancing at the level of a warp of CUDA threads, our runtime demonstrates parts of both of these features with a persistent state on the device that supports more of a streaming and data driven programming model than the launch-wait-relaunch model normally used with CUDA.

Some topics for future work are as follows. At the moment, our runtime handles device memory allocation and transfer for the programmer, but freeing of device memory cannot occur while a kernel is running on the device, and therefore cannot happen while our runtime is being used. Therefore, in order to limit waste of device memory and of page locked host memory our runtime system should include some more advanced memory reuse mechanisms. This may include the implementation of a concurrent memory manager on the device.Some optimization may be possible on our device work stealing code, with a focus on minimizing the use of atomic instructions and memory fences. While these are necessary to ensure each worker has a consistent view of other workers' deques we may be over-using these instructions. We would also like to consider the results of decreasing the number of threads in each worker on the device. Angels et al. used warps as task execution units. Investigating the effect that a change in worker granularity would have on our system could be very beneficial (or

damaging) to overall performance. Finally, we also aim to integrate this work into the larger Habanero-C parallel programming language project [16] at Rice University.

References

1. N. S. C. in Tianjin, "Tianhe-1A," http://www.top500.org/system/details/10587, November 2010.
2. Nvidia, "CUDA," http://developer.nvidia.com/cuda-action-research-apps, 2011.
3. P. Charles *et al.*, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*, NY, USA, 2005, pp. 519–538.
4. M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar, "Cnc-cuda: declarative programming for gpus," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 230–245.
5. H. R. Group, "Habanero c," https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C, 2010.
6. A. Duran *et al.*, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *ICPP'09*, 2009, pp. 124–131.
7. D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, January 2010.
8. "The Java Grande Forum benchmark suite," http://www.epcc.ed.ac.uk/javagrande/javag.html.
9. M. A. Lastras-Montano *et al.*, "Dynamic work scheduling for gpu systems," in *International Workshop of GPUs and Scientific Applications (GPUScA 2010)*, 2010.
10. X10 2.1 CUDA, "http://x10.codehaus.org/x10+2.1+cuda."
11. D. Cederman and P. Tsigas, "On sorting and load balancing on gpus," *SIGARCH Comput. Archit. News*, vol. 36, June 2009.
12. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 863–874.
13. C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, February 2011.
14. E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Springer-Verlag, 2009, pp. 851–862.
15. "Opencl 1.1," http://www.khronos.org/opencl/.
16. Habanero-C, "https://wiki.rice.edu/confluence/display/habanero/habanero-c."