Dynamic Task Parallelism with a GPU Work-Stealing Runtime System, Max Grossman

Problem and Motivation

While CPUs have been at the core of everything from personal computing devices to the largest supercomputers for decades, their general-purpose architecture is poorly suited for many critical problems, including applications in medical imaging, physical simulations, cryptography, and molecular dynamics. The plateauing of clock cycles and drastic increases in power demands of modern processors has served to further limit the application domain in which CPUs are the best choice, and made future homogeneous platforms impractical. These factors have led to an explosion in interest in multicore and heterogeneous computing, evident across the computing scale from desktops to supercomputers and driven by applications whose computational requirements have yet to be met. This interest has led to a search in the parallel computing community for novel approaches to programming heterogeneous platforms, including new programming models and architectures.

Using Graphics Processing Units (GPUs) for general purpose computation provides an immediate solution to the inadequacies of homogeneous platforms, at once meeting the demands for both increased performance and reduced power consumption. As evidence, consider that 3 of the top 5 supercomputers by performance and half of the top 10 energy efficient supercomputers by energy efficiency use GPU coprocessors (April 2012)[2][3]. Huang et al[1] in 2009 demonstrated that the energy efficiency of GPUs is 763 times that of single threaded code and 237 times that of multithreaded code, based on the energy-delay metric which takes both performance and energy consumed into account. GPUs are also widespread in personal computing, present in any laptop or desktop machine with a display and generally restricted to graphics computation. However, GPUs also have a well-deserved reputation for being difficult to develop software on, optimize applications for, and debug.

Before getting into details of this work, it is important to provide a brief background on GPUs, NVIDIA's Compute Unified Device Architecture (CUDA)[4] and a few concepts used in this work. GPUs are many-core processors, containing hundreds of processing units. These processing units are divided among streaming multiprocessors (SMs), of which there are usually 10-20. Each of these SMs has access to global memory on the device, as well as a small amount of memory local to each SM. This work will use CUDA, NVIDIA's data parallel programming model, as a means of implementation. CUDA exposes this processor architecture through collections of SIMD threads, called blocks of threads. A single block of threads is guaranteed to execute on the same SM. In this work, scheduled tasks are actually data parallel pieces of work called blocked tasks which eventually will be executed by a block of CUDA threads. While CUDA was used initially, current work focuses on implementing the concepts described below in OpenCL[11] and on an AMD Accelerated Processing Units (APUs), which combines a multi-core CPU, a GPU, and shared inter-device global memory on a single chip.

There are many difficulties and overheads that arise in general purpose GPU development, as is true for most new architectures and co-processors. Included among these are:

1. Explicit device memory management, including allocation and communication
2. Efficient use of local memory
3. Coalesced memory accesses
4. Stream and event management for GPU-CPU concurrent execution
5. Conditional divergence

This work will focus on enabling a larger set of programmers to quickly develop optimized code on GPUs by easing many of these difficulties, facilitating the adoption of this technology in order to accelerate a wider range of applications.

Approach and Uniqueness

The high level goal of this API and runtime system is to map familiar concepts from task parallel programming models for multi-core CPUs onto the unfamiliar and challenging architecture of many-core SIMD GPUs. This system will demonstrate hybrid parallelism by featuring both data-parallel and task-parallel characteristics. While task scheduling and creation will occur dynamically as in existing task-parallel programming models such as Cilk or X10, the blocked tasks being scheduled are actually data parallel chunks of work that can be efficiently executed by both the CPU and GPU depending on the task block dimensions. This hybrid parallelism will pave the way for a much broader class of applications to take advantage of the GPU's processing power.

Consider an Unbalanced Tree Search (UTS), in which the algorithm explores every node of an unbalanced tree. While not a traditional candidate for GPU execution because of its more task parallel structure where each new node is explored by a dynamically created task, it is a perfect example of how GPUs are applicable to a large field of applications for which they are normally overlooked: tree and graph algorithms. If we closely scrutinize graph search algorithms, they actually demonstrate hybrid parallelism. While exploration of new nodes is done in a task parallel fashion with a new and independent task being created for each node, within nodes there is data parallelism over neighbors of the node. For instance, in the multi-core CPU implementation of the UTS benchmark a new task is created for each new child node being explored, and each of those tasks performs tests on each of its children in data-parallel fashion to determine if they should be explored in turn. As a result, we have an application demonstrating hybrid parallelism which is suitable for execution on the GPU given that dynamic task creation is supported, which this work does. There is therefore a large number of important tree and graph based applications which can benefit from this work facilitating and enabling their porting to GPU execution.

This work approaches the problem of GPU programmability from two directions by building a novel API and runtime using NVIDIA's CUDA programming model.

The high level goal of the API is to provide a simplified programming interface for the GPU to the software developer, using basic function calls which correspond to easily understandable concepts which are familiar from task parallel programming on multicore CPUs. The main functionality provided is through an *insert_task* function and a *get_data* function. *insert_task* can be called from both the host and device and creates new work on the device, which *get_data* allows for retrieval of output from the device

With this API, we accomplish a number of goals. First, all memory allocations and transfers of data between the host and device are handled asynchronously by a part of the runtime which resides on the host. Second, a multi-device abstraction is provided for the user which represents multi and single GPU systems uniformly and distributing work across different devices using a round-robin algorithm. More intelligent placement of tasks based on inter-task dependencies or device utilization is included in future work. Third, the API offers a simplified task parallel user-facing programming model.

The more significant contribution of this work is the runtime, which provides the services that such a simplified API requires. In this work, I developed a persistent, load balancing, hybrid work-sharing and work-stealing runtime system in CUDA for execution on GPU devices which enables a number of features not possible in vanilla CUDA. At its simplest, this runtime is a while loop executing in a block of threads (or worker block) on each SM of a device searching for tasks to execute based on policies which promote data locality:

```
while(quit_loop==0) // while we haven't received kill signal from host
{
    if(threadIdx.x == 0) // only master thread should look for new work
    {
    int success = 0; // have we gotten work?
    success = pop_private(...); // look for local task
    if(success == 0)
    {
        success = pop_shared(...); // look for task from host
         if(success == 0) success = steal(...); // steal from other worker
     }
 }
 __syncthreads(); // wait for master thread here
 if(success) handle_task(task); // run task
}
```

Each worker block on the device owns its own private work-stealing deques which it can insert new work into, and which any block can retrieve work from. These work-stealing deques add significant functionality to the GPU programming model, including:

1. Load balancing across SMs on the same device, as starved SMs steal work from others.
2. Dynamic task creation on the device, a feature not included in CUDA. To a certain extent, this also

enables efficient dynamic memory allocation as each task on the device has a certain chunk of memory associated with it.

3. Automatic management of dependencies between tasks.

In addition, a global work-sharing queue exists which is readable by all worker blocks, but only writeable by the host system. This work-sharing queue permits the streaming of new tasks to the device from the host while the runtime is executing. This relationship between host and device is portrayed in Figure 1.

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.[8] implemented a work stealing runtime on the device. In their paper, a worker is a single warp of threads (32 threads) while in ours a worker is a single block of threads, which can be of varying size. Their use of queues in shared memory would yield less latency than our global memory queues, and could be included in future work. 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 or lead to premature termination.

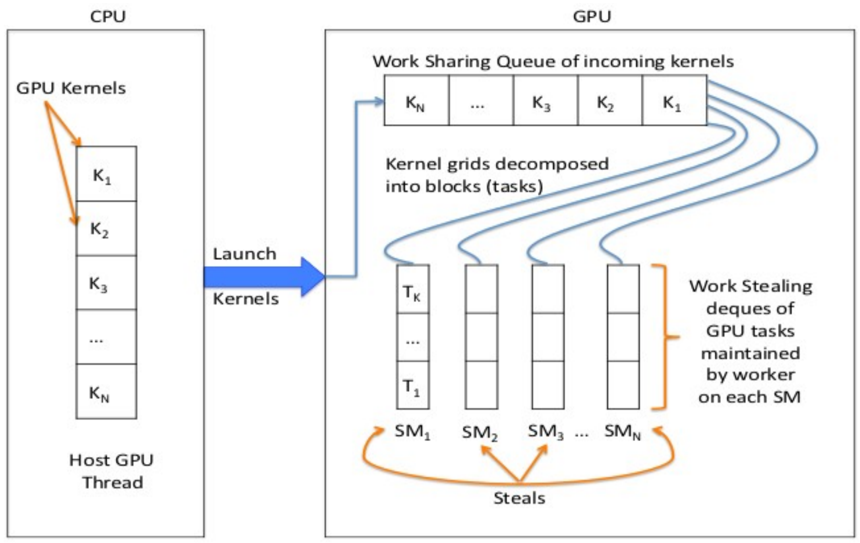
CPU
GPU Kernels
K_1
K_2
K_3
...
K_N
Host GPU Thread
Launch
Kernels
GPU
Work Sharing Queue of incoming kernels
K_N
...
K_3
K_2
K_1
Kernel grids decomposed into blocks (tasks)
T_K
...
T_1
Work Stealing deques of GPU tasks maintained by worker on each SM
SM_1
SM_2
SM_3
...
SM_N
Steals

Figure 1. Depiction of relationship between GPU and CPU, as well as the queue structure on the device.

The X10[9] programming language recently began supporting the execution of tasks on CUDA devices. They do not provide an actual on-GPU work stealing runtime, but instead provide the user with a simpler API for allocating device memory, copying asynchronously and expose the block and thread CUDA memory model as nested loops iterating over X10 points. However, this is not a device runtime and even though 1) the appearance of the code may be more familiar to non-CUDA programmers and 2) 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.

Work by Cederman and Tsigas[10] presents 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. 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.

Results

To test the performance of the runtime, examples of applications were used that are either 1) challenging to implement efficiently on graphics hardware, or 2) data parallel and already well suited for CUDA. This was done to demonstrate good performance with suboptimal algorithms, as well as low overhead. Benchmarks tested include:

1. NQueens from BOTS. Implementation of NQueens exhibits unbalanced computation trees and requires lots of dynamic task creation and load balancing. All of these are features which cause poor performance on GPUs.

2. Quicksort sorting algorithm[6]
3. Crypt and Series benchmarks from the Java Grande Forum Benchmark Suite[7]. Both are regular and recognized to be a good candidate for GPU execution. This benchmark will demonstrate the low overhead of the runtime.
4. Shortest path computation based on Dijkstra’s algorithm. This benchmark starts with a single task and must then spread the load across all SMs as evenly as possible.
5. Unbalanced Tree Search (UTS). UTS, like NQueens, exhibits unbalanced computation trees and requires lots of dynamic task creation and load balancing.

In these tests, runs are executed with different numbers of devices as well as different data sizes to observe the effects on execution time. Additionally, we use diagnostic data from our runtime to measure how effectively the runtime balances the load of the application.

Benchmark tests were performed with 1, 2, or 3 NVIDIA Tesla C2050 GPUs. 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).

Figure 2 records speedup for crypt and series normalized to a hand coded CUDA implementations. With crypt, there is a close relation between hand coded on 1 device and runtime with 1 device, proving the runtime had little overhead there. However, Series demonstrates consistent slowdown across all tests. The cause of this is added overhead from memory transfers which the runtime requires but the hand coded implementation does not because it takes constants as input. Note that the runtime with 2 devices easily outstrips the hand coded version, and also that it is a rare edge case where computation takes only constants as input.

Figure 3 shows the number of tasks executed on each SM in a test run of NQueens and Dijkstra. Despite the fact that these applications have unbalanced static computation trees, there are only small variations in the number of tasks run by each SM. For example, sequential executions of NQueens showed that the distance to the deepest leaf could be 9.8x that of the shallowest leaf in the computation tree. In these tests, the SM with the most tasks only executes 1.6x as many tasks as the SM with the least number of tasks. Additionally, Dijkstra's load balancing has a mean of 53998 tasks per SM with a standard deviation of only 380 tasks.

Figure 4 contains performance data on two task parallel benchmarks that are unsuited for execution on the GPU. As noted above, both UTS and NQueens are characterized by attributes which make them poor candidates for execution on GPUs. Still, for many test cases 2 devices were able to perform on par with 12 host threads running on a 12 core CPU in a highly optimized version of the host code.

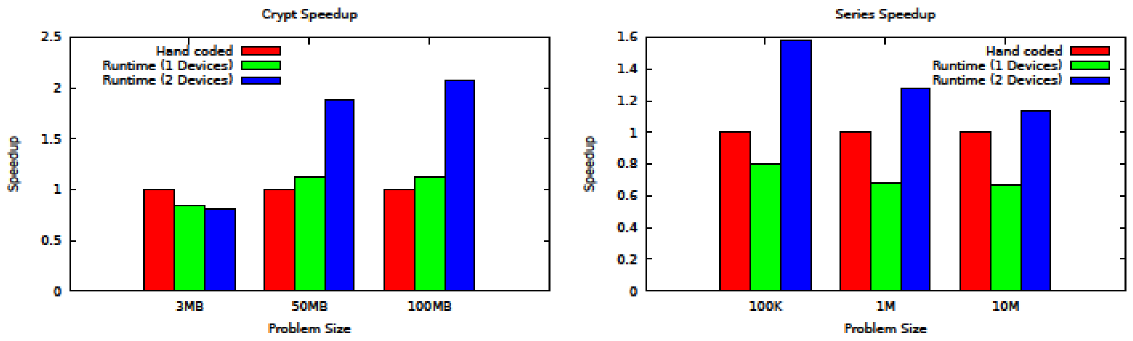

Figure 2. Crypt and Series speedup running with hand coded CUDA, the runtime using 1 device, and the runtime using 2 devices.

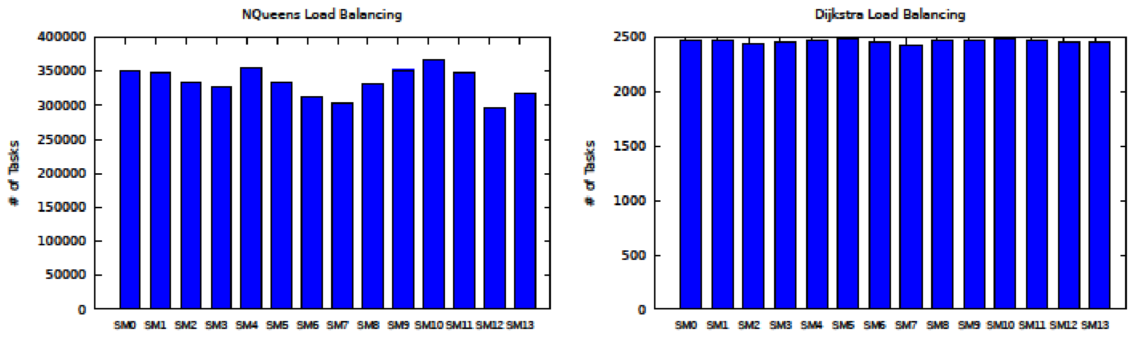

Figure 3. Tasks run by each SM in the NQueens and Dijkstra benchmarks.

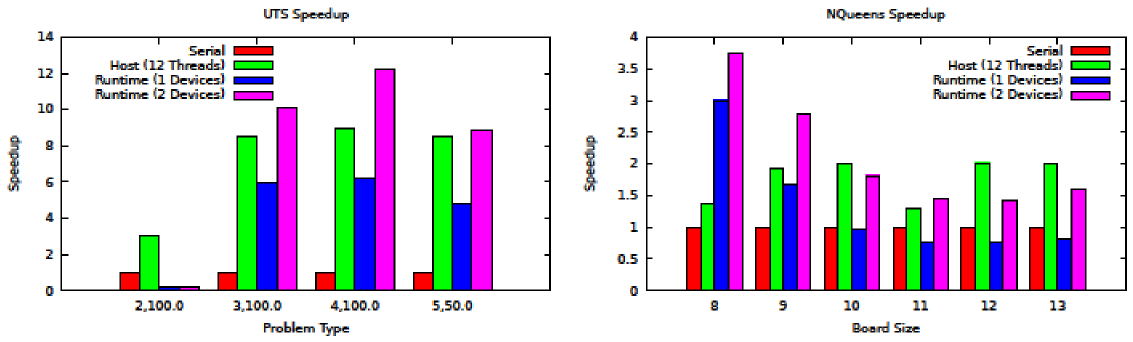

Figure 4. Execution time of UTS and NQueens benchmarks

Conclusions and Contributions

This work demonstrates the first fully-featured, multi-GPU, persistent, load balancing runtime that support s dynamic task parallelism through a hybrid work-stealing and work-sharing device runtime. The effectiveness of the hybrid work stealing and work sharing queues in distributing tasks across the device was shown 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. It was proved that this runtime incurs little overhead for applications already well suited for CUDA (such as Crypt and Series), and that better performance might even be achieved as a result of the provided multi-device management and overlapping of data transfers with kernel execution. Finally, the UTS and NQueens benchmarks also showed the extension in GPU capabilities to task parallel, irregular applications which graphics hardware is not normally targeted at and the potential for performance improvements there. A short description of a drastically simpified API for graphics hardware was also given.

The impact of this work is two-fold. First, this demonstrates that perceptions about the applications domain of GPUs may need to be re-evaluated. Even unoptimized kernels were able to outperform highly optimized multithreaded host implementations of irregular applications. Using this device runtime makes excellent performance much more achievable for these types of applications. Second, this work can also help to accelerate the development of performance critical code by providing a simpler programming interface to a powerful piece of hardware for developers who are not experts in parallel computing.

References

1. S. Huang, S. Xiao, W. Feng. "On the energy efficiency of Graphics Processing Units for Scientific Computing". http://accel.cs.vt.edu/sites/default/files/paper/huang-hppac09-gpu.pdf

2. Top 500 Supercomputers. http://www.top500.org/

3. Green 500. http://www.green500.org/

4. NVIDIA, “CUDA,” http://developer.nvidia.com/cuda-action-research-apps, 2011.

5. M. Grossman, A. S. Sbırlea, Z. Budimli´, 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.
6. D. Cederman and P. Tsigas, "GPU-Quicksort: A practical quicksort algorithm for graphics processors," J. Exp. Algorithmics, vol. 14, January 2010.
7. "The Java Grande Forum benchmark suite," http://www.epcc.ed.ac.uk/javagrande/javag.html.
8. M. A. Lastras-Montano et al., "Dynamic work scheduling for GPU systems," in International Workshop of GPUs and Scientific Applications (GPUScA 2010), 2010.
9. X10 2.1 CUDA, "http://x10.codehaus.org/x10+2.1+cuda."
10. D. Cederman and P. Tsigas, "On sorting and load balancing on gpus," SIGARCH Comput. Archit. News, vol. 36, June 2009.
11. "OpenCL 1.1," http://www.khronos.org/opencl/.