© 2008 Lukasz Wesolowski

AN APPLICATION PROGRAMMING INTERFACE FOR GENERAL PURPOSE GRAPHICS PROCESSING UNITS IN AN ASYNCHRONOUS RUNTIME SYSTEM

BY

LUKASZ WESOLOWSKI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Adviser:

Professor Laxmikant V. Kalé

Abstract

Graphics processing units today are more than a hundred times faster at executing floating point operations than the fastest available multicore processors. While systems for general purpose programming on the GPU are becoming available, many programming tasks on the GPU remain difficult today as a result of the relative immaturity of these solutions. In particular, tools for using a large number of GPUs as co-processors in clusters and supercomputers are lacking.

In this work we present an extension to NVIDIA's Compute Unified Device Architecture which enables writing parallel Charm++ applications for execution on a large number of hybrid CPU/GPU nodes. We demonstrate effective usage of this library by modifying ChaNGa, an existing Charm++ application for cosmological simulations. We also propose an alernative methodology to allow for fine-grained management of work on the GPU.

Table of Contents

1 Introduction

Despite a persistent exponential increase over time in the performance of CPUs, the insatiable demand for fast calculation in areas such as physical and molecular simulation, financial modeling, and weather forecasting continues to require the use of supercomputers to speedup computation beyond what is attainable using traditional PCs. System manufacturers have typically accomplished speedups of up to a hundred thousand over the fastest available PCs by using racks of processing nodes and complex interconnect systems. The IBM BlueGene series, one of the most successful such system to date, allows for the use of hundreds of thousands of processors concurrently. The largest BlueGene/L configuration to date has a peak computational capability of almost 600 TFlops [reference].

Even with the success of supercomputing systems based on general purpose CPUs, there exists a possibility to further push current technology limits and achieve significantly more impressive performance. The key observation is that current generation CPUs do not represent the most efficient use of silicon area for floating point computation. Traditionally, CPUs are general processing devices, which execute a small number of different instructions concurrently, with large parts of the chip area spent for cache and other optimizations used to hide memory latency [data for

The idea of using custom chips to speed-up floating point computation is not a new one, but in the past it has been difficult to realize. Since custom chips are generally built on older process technology, and development takes several years, it has frequently occurred that by the time such chips were manufactured, general CPUs could perform the same computation faster and at a fraction of the cost

[example]. The alternative to custom chips is to use existing computational devices, possibly as co-processing units. The most attractive such option today is the general purpose GPU.

The advent of powerful graphics processing units within the past decade has provided consumers with devices having a hundred times higher peek floating point performance than the fastest available CPUs [reference]. Until recently, it has been difficult to use GPUs for anything other than graphics applications. Developers trying to employ the GPU in a different problem area had to map the operations performed by their algorithm into the small number of fast primitives for shading and rasterization available in the API. In recent years, manufacturers of graphics cards have begun promoting the use of GPUs for general purpose computation. Towards this goal, they have added special hardware on the GPU die, along with software tools that provide a clean general purpose programming interface.

The term general purpose GPU is somewhat misleading, since effectively using the device requires a massive number of threads executing in single program, multiple datastream (SPMD) fashion and extensive data parallelism in the application. Most applications written for supercomputers already satisfy the latter characteristic; this fact potentially makes the GPGPU an interesting option for supercomputers. However, due to the inefficiency of the GPU in performing some types of computation, it is unlikely that GPUs or similar devices will replace CPUs entirely in future supercomputers. More likely is the use of GPUS in heterogeneous systems, where they will be used as coprocessors, and possibly only in a fraction of the nodes.

Current, officially supported APIs for GPGPUs are not optimized for usage in large-scale heterogeneous systems utilizing an asynchronous runtime system. The first fundamental flaw of these software packages is that they do not provide a clean interface to allow treating the streaming multiprocessors on the GPU as separate devices. Instead, to get good performance on the GPU, a single GPU kernel needs

to have enough parallelism to allow all processing units to execute simultaneously and hide memory latency overhead. There are no facilities for executing a piece of code on a subset of the execution units, and allocating additional execution units on demand as more work becomes available.

The second serious problem with the official programming APIs for GPUs is that they provide only a rudimentary mechanism for managing execution of kernels on the GPU. Kernels are spawned and GPU memory copies are initiated from the CPU, so that the CPU is tightly involved in the process of executing code on the GPU. The need to manage GPU kernels on the CPU is not a major nuisance when executing GPU-bound programs which make minimal use of the CPU for computation. When writing a large-scale application for a heterogenous supercomputer, on the other hand, more sophisticated GPU management tools are desirable to simplify coding and improve performance.

In this thesis we propose solutions built on top of CUDA to address the above issues. Our implementation is embedded within the Charm++ asynchronous runtime system to provide a comprehensive framework for programming large scale supercomputing applications for hybrid CPU/GPU systems.

The rest of this work is organized as follows. Chapter 2 contains an overview of current generation GPUs, both in terms of the hardware and the general-purpose programming software packages. In Chapter 3 we review Charm++ and present the features which make it fitting as a host system for our API. In Chapter 4 we introduce Hybrid API and compare it to using CUDA directly. Chapter 5 examines the overhead of using Hybrid API. Chapter 6 presents an example of using our API to accelerate the cosmological simulator ChaNGa. Chapter 7 presents a set of planned extensions to the API, and Chapter 8 contains future work and some closing thoughts.

2 Graphics Processing Units

The first graphics processing units were developed in the early 1980s to accelerate graphics rendering in workstations and personal computers. Early GPUs were often modified versions of existing CPUs specialized for accelerating rendering of 2D bitmapped grapics. The situation changed in the mid 1990s, when the rise of 3D graphics applications, especially games, marked a return to specialized chips for acceleration of 3D graphics. Since then, the demand for increasing levels of realism in graphics has led graphics card manufacturers to design complex devices capable of fast floating point computation, albeit specialized for the types of operations (shading, rasterization, filtering) required for graphics.

By 2003, GPUs became faster than CPUs at executing single-precision floating point operations. The gap between GPU and CPU floating point computation rates has been growing increasingly large since then. The fastest GPU today, the NVIDIA GeForce GTX 280, computes up to 933 billion floating point operations per second (GFLOPS), compared to 50 GFLOPS in the fastest available CPUs [reference].

2.1 GPU and CPU Fundmantals

There are several reasons for the disparity between GPU and CPU floating point performance. First, CPUs and GPUs have very different application areas. CPUs are general purpose units designed to perform computation and manage devices in desktop, server, and notebook computers. Traditionally, fulfilling this purpose required efficient execution of one program thread at a time on a single execution core.

This model changed to some degree as computer usage patterns shifted to execution of several desktop applications simultaneously on a single machine. CPU designs thus moved first toward concurrent execution of multiple threads via simultaneous multithreading, and, more recently, toward multicore designs. Even so, desktop applications remain overwhelmingly single-threaded, and it is yet to be determined whether typical users can make good use of 16 or 32 cores.

Regardless of the number of cores it has, a CPU must be able to execute a single memory-intensive thread very fast. Today, this typically means allocating large parts of the chip area for cache memory and optimization logic such as branch prediction/resolution units.

Unlike CPUs, GPUs have until recently been developed with a single narrow purpose in mind – the rendering of graphics. This task can be parallelized efficiently, so that additional processing units can be readily utilized to accelerate execution.

The effectiveness of adding additional execution units on graphics chips has encouraged the design of increasingly large chips. As a result, GPUs quickly progressed to become massively parallel machines. GPUs today have a significantly larger chip area than CPUs. The latest GT200 GPU chip from NVIDIA has roughly 1.4 billion transistors, as compared to about 400 million transistors in Penryn family quad-core Intel CPUs. In addition, due to the particular nature of graphics rendering, GPUs have no need for branch prediction and other types of optimization logic normally found in CPUs.

If GPUs do not have the kind of hardware that allows CPUs to execute single-threaded applications very fast, we may wonder what it means for a GPU to be identified as "general purpose." In order to answer that question, it is instructive to take a look at how GPGPUs came to be.

2.2 The General Purpose GPU

Since the early days of GPUs, researchers have experimented with using the devices for accelerating applications unrelated to graphics. These efforts intensified as GPUs became faster than CPUs at performing single precision floating point operations [references]. Writing general purpose GPU applications in those days amounted to mapping the desired operations onto the data structures and functions available in the graphics API. This kind of work, while very tedious, could lead to significantly better performance than what could be achieved on the CPU.

Graphics card manufacturers soon realized that they could market and sell their products as coprocessors for fast computation. In order to increase the adoption of this approach, a user friendly general purpose programming interface was needed. AMD and NVIDIA, who by this point controlled most of the market for GPUs, developed separate software for this purpose. NVIDIA named its solution Compute Unified Device Architecture, or CUDA, while AMD came up with a low-level interface called Close to Metal (CTM), and more recently with Brook+, an extended version of an existing high-level GPGPU language.

2.3 GPU Architecture

The GPU industry today is dominated by two players: ATI, currently a subsidiary of AMD, and NVIDIA. Examining the chips produced by these two companies gives one a good sense of the current state of GPU architecture.

At the heart of current generation NVIDIA GPUs are hundreds of simple execution units for shading. These processing units are homogeneous and scalar, as distinct from separate vertex and pixel shading units found in past generations of GPUs. The unified shader design leads to better load balancing for graphics rendering, and the scalar nature of these units decreases scheduling complexity and allows

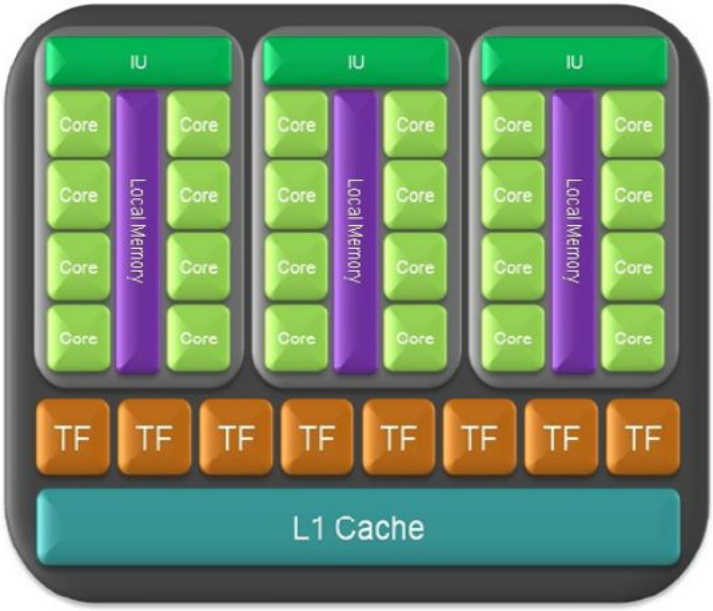

Figure 2.1: The Texture Processing Cluster in the GT200 GPU

for higher clock rates. This type of architecture, aside from being useful for graphics, is also desirable for a general purpose API, since it removes the need to handle multiple types of execution units in differing ways, along with the load balancing issues which that would present.

The shader units in NVIDIA GPUs are called streaming processors (SPs). SPs are organized into groups of eight known as streaming multiprocessors (SMs). Each SM has its own register file, instruction/constant cache, and issue/control logic. An interesting feature of the architecture is a small amount of shared memory per SM apparently added to improve the performance of the device for general purpose programming using CUDA. SMs are grouped along with texture addressing and filtering hardware into units called texture processing clusters, or TPCs. Figure **??** shows a block diagram for a TPC. A group of raster operators provides the means for outputting pixels, as well as an interface to the global memory on the graphics card.

The latest GT200 GPU from NVIDIA contains 10 TPCs, each with three SMs,

for a total of 240 streaming processors. Each SM contains 16384 32-bit registers, and 16 KB of shared memory. The chip contains 8 raster operators which provide a 512-bit wide path to the global memory on the graphics card. Up to 1 GB of global memory is available in current generation cards. Peak memory bandwidth is 141.7 GB/s.

Since our API is built on top of CUDA, which is currently supported only in NVIDIA cards, we won't go into the details of GPUs manufactured by ATI. A closer look at these chips would reveal that they contain similar components, with some differences in the component organization. As an example, RV770, the latest generation GPU from ATI, has 800 stream processors, but they are organized in groups of 5 into superscalar units controlled by very long instruction word (VLIW) commands. These units have about half the clock rate of the shader units in the NVIDIA design, which follows from the greater complexity of the VLIW design. A further consequence of the VLIW design is the necessity for a more complex scheduler to ensure the superscalar units execute at peak capacity most of the time.

2.4 Memory Considerations for Efficient Execution

There are several architecture-specific requirements which a piece of code executing on the GPU must satisfy to yield good performance. First, the program needs to have a high ratio of computation to memory access operations. This is a result of the order of magnitude difference between the peak performance of the execution units and the memory bandwidth. Furthermore, since the number of execution units in GPU designs over time increases faster than the memory bandwidth, the ratio of computation to communication in GPU applications needs to increase across generations of GPUs to maintain good performance. This requirement is easily met

in graphics, where demand for higher levels of realism requires increasingly more vertex and pixel shader processing.

Another memory consideration unique to GPUs is the advantage gained from accessing memory sequentially. This is an optimization designed to improve rendering performance. Since rendering involves sequential processing of arrays of pixels and vertices, GPUs contain special hardware which optimizes sequential streaming of data to and from memory. While it may not always be possible to take advantage of this feature in general purpose applications, the drastic improvement in memory performance which it provides makes it a good feature to keep in mind for when an opportunity arises to use it.

In subsequent sections we will see general purpose programming paradigms which encourage writing code that takes advantage of the peculiarities in GPU memory performance.

2.5 General Purpose Application Programming Interfaces

Before the first general purpose APIs were developed for GPUs, those who were eager to utilize GPUs for applications other than graphics had to use either assembly level programming or special shading languages for this purpose. Shading languages, including GLSL, Cg, and HLSL, were developed to abstract graphics hardware and give developers of real-time graphics applications greater control over the rendering process. Using shading languages for general-purpose work was an arduous task which required packing data into textures and manipulating these via shading operations.

While shading languages were making an impact in the industry, researchers in academia were investigating high-level paradigms for general purpose computa-

tion. The most promising direction in these efforts was the *streaming* approach to programming the GPU.

2.5.1 Stream Programming

The stream programming paradigm defines computation in terms of operations performed on a set of data elements, or *streams*. In this context the set of performed operations is called a *kernel*. The elements in the stream execute the kernel independently, and the system writes the results to an output stream, with one output element per input data element.

The streaming paradigm fits well with the GPU model of execution. A program which uses the GPU for rendering graphics can be interpreted as a kernel executing on streams of vertex and pixel data.

Streaming presents several advantages over other paradigms in the context of GPU computing. Independent execution of the kernel allows the data elements to be processed in parallel and simplifies control logic. If the number of elements in the stream is much greater than the number of shader execution units on the chip, this approach also allows hiding the memory latency by suspending elements which are waiting for data to arrive from memory and allowing other elements to proceed with computation.

Since streams will normally be organized linearly in memory, streaming encourages sequential memory access, which, as described above, gives an order of magnitude improvement in performance compared to random memory access.

It is important to keep in mind the advantages and limitations of the streaming approach for general purpose programming. On one hand stream computing allows a programmer to harness the power of hundreds or thousands of execution units concurrently while hiding memory latency costs. On the other hand, effective usage of the model requires large data domains with elements that can benefit from

concurrent independent execution.

2.5.2 Brook

Brook was one of the earliest and most influential GPU stream programming languages [reference]. In its original form it was a language for experimental streaming computers. It was ported to the GPU as it became apparent that the programming model was a good fit for the GPU. Brook formalized the notion of a stream into a programming construct. Kernels in Brook are functions which operate on data streams of three different types: in, out, and gather. In(put) streams are read-only. For any instruction within a kernel involving an *in* stream, the system executes the instruction on every element of the stream. Output is written to the stream specified with the keyword *out*, which could be the same as the input stream. Gather streams allow arbitrary indexing of stream elements. While this improves the generality of the model, gather streams do not take advantage of the improvement in bandwidth from sequential memory access. Other influential features provided by Brook are reduction constructs for streams, variable output streams, and scatter, gather operations.

One notable successor to Brook is Brook+, a new high-level API for GPUs currently in development at AMD.

2.5.3 CUDA

Brook also influenced the development of NVIDIA's Compute Unified Device Architecture(CUDA), perhaps the most popular langugage for general purpose programming on GPUs currently in use. Only NVIDIA graphics cards currently support CUDA.

CUDA borrows many features from Brook, but there are also significant differ-

ences between the two systems. One of the main goals in the design of CUDA was to make its syntax appear as close to C as possible. As a result, CUDA does not include special constructs for streams. Instead, in CUDA the programmer passes arrays as input to the kernel. Device arrays in CUDA can be accessed via arbitrary indexing, making them similar to gather streams in Brook.

In place of streams, CUDA introduces the notion of *threads* and *blocks*. Threads are single units within the larger structure of a kernel instance. A block is a 2 or 3 dimensional array of threads. A kernel instance consists of an array of blocks. Every thread in a kernel executes the body of the kernel. Each thread contains an index within its block, and every block contains an index within the kernel - these values are available inside the kernel to allow each thread to perform operations on different data. A group of 32 consecutive threads is called a *warp*.

When a kernel is executed, the system assigns full blocks of threads to each streaming multiprocessor. There is a limit both in terms of the maximum number of blocks and threads that can be assigned to an SM. Any remaining unassigned blocks are assigned later on as previous blocks finish execution.

CUDA does not provide any guarantees about which blocks will execute simultaenously, and which blocks will occupy the same SM. In fact, threads in different blocks cannot be synchronized, and the only synchronization call synchronizes all threads in a block.

Within each SM there are 8 streaming processors. The execution model involves iterating over the warps residing on each streaming processor, and executing the kernel on the warp exclusively before switching to a different warp in the same block. Since a warp consists of 32 threads, this is done by executing consecutive groups of 8 threads every cycle. Eventually, the system switches to executing a different block.

Since up to 768 threads may reside on an SM at any given time, the above

mechanism allows for tolerating memory latency overhead by swapping out warps having threads with pending memory loads/stores, and alloowing other warps to execute on the streaming processors in the meanwhile. It is instructive to compare this approach to CPUs, where memory latencies are hidden primarily by using caching. The advantage of the GPU approach is that it allows the usage of more chip area for execution units. The disadvantage is that it requires massive data parallelism.

To utilize the improved bandwidth of sequential memory access in CUDA, a programmer needs to write his code so that data in arrays is accessed sequentially. One common scenario uses the thread id of each thread as an index into the input and output arrays. This essentially mirrors the streams in Brook. In applications where memory bandwidth becomes a limitation, the programmer can try utilizing the special on-chip shared memory. Shared memory is about as fast as the register file, but there are several caveats the programmer needs to be aware of to ensure good performance. First, shared memory is very small (only 16 KB on the G80), so its scope is limited when it is being shared by hundreds of threads in a block. Secondly, shared memory is banked (16-way on G80), so to get good performance, each thread in a half-warp needs to access data in a different bank.

CUDA requires the programmer to be aware of many architectural details. As an example, every thread consumes a set of registers in the register file. Registers are occupied on the granularity level of a block. When a block is swapped out and control is transferred to a new block, the register file is saved for later execution. If too many threads are assigned per block, it may be that the register file will not be capable of holding all the required data, and the program will crash. It is the programmer's responsibility to ensure this does not happen.

Branches are handled by creating a mask representing the threads which take the branch, and then executing both directions in the branch on the whole warp

using the mask. The only way to avoid this behavior is to write applications in such a way that all threads in a warp follow the same direction in the branch. This is why programs with complicated control flow will usually give poor performance in CUDA.

In chapter 4 we will demonstrate extensions to make CUDA more applicable for use in Charm++, a parallel C++ library with an asynchronous runtime system. We will now look at Charm++ and its features.

3 Charm++

Charm++ [reference] is an object-oriented parallel language based on C++ and the idea of object-based virtualization. In Charm++ we view the program as a collection of objects which execute code in response to messages received from other objects, and in turn send messages of their own. It is the programmer's responsibility to partition the application in this fashion. The Charm++ runtime system takes the virtual representation of the application and maps the objects to physical processors. If there are more objects than physical processors, the runtime system will map multiple objects per processor. Charm++ supports grouping objects into indexed collections of multiple dimensions.

Objects in Charm++ communicate through asynchronous method invocations, or messages for short. The runtime system handles transmitting messages between objects by keeping track of the location of each object. The runtime system enqueues messages on the destination processor. When a processor becomes idle, the runtime system executes a method by taking a message from the front of its queue. Typically the method executes without interruption until it finishes, although if the method is labelled by the programmer as threaded, the runtime system is free to interrupt it and schedule it for later execution. Messages which need to be executed as soon as possible can be prioritized to avoid long delays in the queue. Other than simple point-to-point messages, Charm++ supports collective communication operations, including broadcasts, reductions and multicasts.

Charm++ applications are typically written to have significantly more objects than the number of physical processors used during execution. The presence of mul-

tiple objects on a single processor allows for automatic overlap between computation and communication. If one object is waiting for a message, another object can execute in the meanwhile, ensuring the CPU does not become idle. It is instructive to notice that this is the same basic mechanism that allows GPU threads in CUDA to overlap computation with memory access

CHARM++ provides an intricate load balancing framework. The runtime system includes an instrumentation mechanism which records the computation and communication loads for each object based on transmitted massages. Since parallel applications are typically iterative, the runtime system can use this information to re-map the objects to processors in such a way as to achieve better load balancing. A number of strategies are available to determine how the runtime system will perform the remapping. Which strategy works best is highly application-dependent. As a result, the choice of the strategy is left to the user. Instrumentation can be disabled by the user for a slight improvement in performance in applications which do not require load balancing.

4 Hybrid API

In this section we present the Hybrid Application Programming Interface, an extension to CUDA which provides a convenient mechanism for using CUDA-enabled GPU to write *hybrid* CPU/GPU parallel applications. While using CUDA directly in Charm++ is possible, our API provides abstractions to automatically manage the execution of multiple kernels, freeing the programmer from the responsibility to synchronize after every kernel.

4.1 Using CUDA in Charm++

To motivate the need for a kernel management API, we first present a discussion of the difficulties of using CUDA in standalone mode in Charm++. For the sake of simplicity, we frame our discussion in terms of a fictional Charm++ application. We expect actual applications to follow the same patterns.

Our example application consists of a large number of parallel objects executing on a group of hybrid CPU/GPU nodes. For this discussion, we focus on the objects assigned to a single processor p. Some of the objects on processor p can benefit from executing on the GPU. For the sake of generality, we assume that the GPU-bound part of the computation for these objects consists of several kernels. Using multiple kernels executing in sequence is in fact a common technique to overcome the inability to communicate between threads in different blocks of a kernel grid.

Suppose we are assigned the task of writing this application, using CUDA libraries for kernel management. Before executing each kernel, we will need to copy

a set of input arrays from CPU memory to GPU memory. Secondly, we will need to manage the execution of each kernel. CUDA allows two methods of doing this. The first method blocks on the CPU, waiting for the GPU kernel to finish. Using this operation, we could simply execute the kernels in sequence one after another. Furthermore, we could perform the necessary memory transfer operations before and after each kernel with ease, since presumably we would be operating from within a method of the object which has access to the required data.

(Consider putting figure here showing the execution of the kernels)

The problem with the above approach is that the CPU is left idle every time the GPU executes a kernel. This may not be an issue for GPU-bound applications which use the CPU solely to control the device, but for hybrid CPU-GPU applications it is imperative that the two devices be able to execute concurrently as long as there are no dependencies in the code they execute.

This leads us to the second method for synchronizing kernel execution - polling for kernel completion. In this approach, we can execute CPU code immediately after making a kernel call. Subsequently, we can use CUDA's polling function to periodically check if the GPU kernel has completed execution. Once we detect kernel completion, we can then perform the necessary memory transfers and start the execution of a new kernel. This approach leads to much better hardware utilization in hybrid applications. As long as we have work for both the CPU and the GPU, the two devices will execute concurrently most of the time.

There are two caveats to the polling approach. First, the GPU will be idle in the time after a kernel completes execution and before this is detected by polling on the CPU. This suggests that to minimize GPU idle time, one should poll frequently. On the other hand, polling too frequently decreases readability of the code, and the polling operation itself carries an overhead which wastes CPU time every time we poll while the GPU kernel is still running.

The second problem with polling in this context is specific to object oriented designs. After calling a kernel from one object A, the CPU code we subsequently execute may be in the context of a different object B. If we detect kernel completion from within one of B's functions, we will need to transfer control to A, possibly with a callback mechanism to allow returning the control back to the place where our CPU work was interrupted. This kind of design would muddle the control flow of the application and lead to confusing, difficult to understand code. In a parallel Charm++ application where objects may reside on different processors, the situation is even worse, as invoking B's method from A will likely require sending a message. This will lead to the additional overhead of handling the message in the runtime system, and likely, transmitting the message to a different processor.

Another difficulty in using the above approach with Charm++ is that control flow is often imprecise in Charm++ applications, so that it may be difficult to tell which object will execute after A. The difficulty of using the polling approach directly makes it an unlikely choice for complex hybrid applications.

4.2 A Simpler Approach to Kernel Management

So far we have looked at two methods of writing a parallel hybrid application. The first, synchronous, method is easy to program and produces lucid code, but gives poor performance. The second method can give good performance, but is tedious to program and produces confusing, difficult to maintain code.

The synchronous method is easy to use, because it allows the programmer to perform all GPU operations at once from within a method of the appropriate object. By the time we exit the scope of this object, the whole of the GPU computation for that object is complete, and we need not worry about it anymore.

To provide similar functionality in an asynchronous setting, Hybrid API provides

a mechanism to schedule kernel requests for future execution. This allows the programmer to schedule multiple kernels for a particular object. As in the synchronous approach, the programmer need not be concerned about these kernels outside the scope of the object.

Along with each kernel request, the user needs to specify a pointer to a callback function which the system will invoke once the kernel finishes execution. This function, implemented by the user, will copy data back to the CPU. If needed, the function can then schedule another kernel on the GPU or even call a CPU function dependent on the GPU output.

Note that with this approach we get a reasonably clean control flow and good performance. The user no longer needs to poll to check for kernel completion, leading to cleaner code. Likewise, in situations where multiple kernels can be scheduled simultaneously or through kernel callback functions, there is no need to send messages later on to invoke these kernels, since the system will take care of this. The system also minimizes GPU idle time by polling for kernel completion at semi-regular intervals. This will further be explained in the implementation section.

4.3 Work Requests

From the user's point of view, Hybrid API consists of work requests to define units of execution on the GPU, and a queue for scheduling these requests. A work request contains the necessary paremeters for CUDA kernel execution along with some additional members for automating data transfer between the CPU and GPU. A work request consists of the following data members:

- **dim3 dimGrid** - a triple which defines the grid structure for the kernel; dimGrid.x and gimGrid.y specify the number of blocks in two dimensions, dimGrid.z is unused

- **dim3 dimBlock** - a triple defining each block's structure; specifies the number of threads in up to three dimensions
- **int smemSize** - the number of bytes in shared memory to be dynamically allocated per block for kernel execution
- **void *readWriteHostPtr** - a pointer to a buffer containing user data needed for GPU execution; the system will copy the buffer to GPU global memory prior to execution of the kernel; upon completion of the kernel, this buffer will be overwritten by the buffer in readWriteDevicePtr
- **void *readWriteDevicePtr** - the corresponding pointer in device (GPU) memory; allocated and defined by the system prior to kernel execution
- **int readWriteLen** - the size of each read/write buffer in bytes
- **void *readOnlyHostPtr** - analogous to *readWriteHostPtr, but with the difference that this buffer is not overwritten upon kernel completion
- **void *readOnlyDevicePtr** - analogous to *readWriteDevicePtr, but does not get copied to CPU memory after kernel is finished
- **int readonlyLen**
- **void *writeOnlyHostPtr** - allocated by the user prior to submitting a work request; upon kernel completion, system will write data from writeOnlyDevicePtr here
- **void *writeOnlyDevicePtr** - an output buffer on the GPU
- **int writeOnlyLen**
- **void (*callbackFn)()** - a callback function specified by the user; executed after the kernel and memory transfers have finished

- **int executing** - a flag to help the system keep track of whether a work request has started executing
- **int id** - a kernel identifier supplied by the user to allow the system to call the correct kernel
- **cudaEvent_t completionEvent** - the mechanism for polling in CUDA; allows the system to determine when a kernel finishes execution

Two items in the above list, id and completionEvent, deserve further discussion. CUDA events and completionEvent in particular will be described in the implementation section. But first, we focus on id and the mechanism for kernel definition in Hybrid API.

4.4 Kernels

Kernels in Hybrid API are defined as in CUDA, with a few specific rules which the user needs to follow to allow the automatic buffer transfer performed by the system to work correctly. The first thing the user needs to keep in mind is that parameters to kernels are limited to the data members of the work request corresponding to a kernel. While this may lead to some inconvenience, it is a necessary consequence of the automatic nature of kernel managemement in Hybrid API. Automatic buffer transfer performed by the system is limited to the buffers defined in the work request, and follows the behavior outlined above.

As part of the work request preparation step, the user will need to organize input data into read-only and read-write arrays. The kernel can also write to a third, write-only array. All these arrays have sizes specified by the user when setting up the work request.

To schedule a work request for execution, the user enqueues it to a system GPU

queue. The queue is part of the Charm++ runtime system instance, and so there is one queue for every processor running the Charm++ application. The user needs to declare the queue as an extern variable in the CUDA-specific code. This is done with the command

```
extern workRequestQueue* wrQueue;
```

The user also needs to define the function void kernelSelect(workRequest *wr), which the system calls to execute the kernel for a particular work request. A standard implementation of the function uses a switch statement controlled by the *id* data member of a work request. The kernel calls itself follows for each case of the switch statement, and follows the typical CUDA kernel call syntax.

4.5 Compilation

While it would be convenient to be able to write CUDA and Charm++ code within the same source file, this feature is not currently supported. CUDA and Charm++ require different compilers, making it non-trivial to provide this feature. As a result, the user needs to organize CUDA and Charm++ code in separate files. Naturally, the user also needs to keep in mind peculiarities of the CUDA programming model, such as its lack of support for C++ features, including objects.

4.6 Implementation

Apart from the work request structure presented before, the source code for Hybrid API consists of a queue data structure , functions to manage kernel execution, and an interface into the Charm++ runtime system. The work request queue is a straight-forward C implementation of a first-in-first-out (FIFO) queue.

One aspect of the implementation which warrants some discussion is the interface to the Charm++ runtime system. Recall that Charm++ employs a message-driven programming model, with a runtime system scheduler which gets invoked on a processor every time a method finishes execution. Under typical CPU-only execution, the scheduler examines the queue of incoming messages and selects one based on location in the queue and priority. In our implementation, the scheduler is also programmed to call a GPU progress function. Since querying for kernel completion may be a high-latency operation, the GPU progress function is normally invoked periodically after a set number of scheduler invocations. The exception is when the scheduler determines that there are no messages to process on the CPU. In that case the GPU progress function is called every time, since the CPU will likely be idle anyway.

The GPU progress function *gpuProgressFn* has a simple structure. If the queue is non-empty, it determines whether the work request at the front of the queue is executing. If it is not, gpuProgressFn invokes a function for setting up the data buffers on the GPU, and then executes the kernel by making a call to the *kernelSelect* function, which we have described previously. The function also sets the *executing* flag for the request.

If the executing flag is set for the work request at the front of the queue, gpuProgressFn queries the GPU checking if the kernel is indeed still executing. If the result of the query indicates that the kernel is finished, we invoke a function for buffer transfer to the CPU, and then call the user-specified callback function for the work request. Since the GPU is now idle, we then take the next work request in the queue, set it up for execution on the GPU, and then start the kernel.

5 Asynchronous API

While Hybrid API allows combining Charm++ with CUDA to create applications for hybrid CPU/GPU systems, using the GPU effectively with Charm++ applications may be a challenge due to granularity differences between kernels and CPU functions. Typical Charm++ functions execute for up to several microseconds and involve a single thread running on a single CPU core. GPU kernels, on the other hand, typically involve thousands of threads running on hundreds of execution units. The GPU kernel invocation overhead itself is on the order of tens of microseconds. Finding enough work in a Charm++ object to make GPU execution worthwhile may thus be a challenge. On the other hand, if the work requests of multiple objects could be combined to execute simultaneously on the GPU, the GPU could become a much better fit for the hybrid model.

Simultaneous execution of multiple kernels is not currently supported in CUDA. As a result, we have to be creative to express concurrency of work requests. One solution is to keep the programming model the same, but encourage users to be more clever in the application design. The user could, for example, write code to collect data from multiple parallel objects in order to create a work request with enough data parallelism to utilize the GPU more effectively. The callback function for this kernel could then send messages to the contributing objects with the results of the computation. The effectiveness of this approach is highly application dependent. If the objects are instances of the same class, as is often the case in Charm++, or if they are similar in some way so that their input data can be effectively combined for the GPU kernel, then this approach should be relatively straight forward. The greater

the difference between the objects contributing toward the kernel, however, the more effort and ingenuity the programmer would need to put in to make this work. Since one of our main goals is to keep the already complex CUDA programming model from getting any worse in a hybrid setting, it is undesirable to have this as the only available solution.

Another possible solution is to have the system combine work requests dynamically to form larger kernels. This approach would be difficult to implement, and would most likely require some help from the user in terms of additional information submitted with the work request. It is also questionable how well the system could perform this type of work, at least in the general case where different work requests may have little in common. A specialized solution for cases with large numbers of similar (or preferably identical) work requests would probably be reasonable to implement and could in practice yield good results.

5.1 A Model for Asynchronous Execution

Of the two methods examined thus far, the first one is too demanding of the user, and the second is difficult to generalize effectively. We can arrive at a potentially more effective solution by taking a fresh look at GPU architecture. Most GPU programming models and CUDA in particular treat the device as a single computing engine, but GPUs today are composed of hundreds of execution units, grouped into structures containing register files and special purpose memory units. Processing units in NVIDIA GPUs, for example, are organized in groups of eight in this fashion into Streaming Multiprocessors (SMs). If we could provide an interface to allow treating the SMs as separate devices, we would greatly reduce the granularity gap between CUDA kernels and Charm++ functions.

Another advantage of this approach is to reduce the amount of time work requests

spend in the queue. Under this scheme, as long as some of the GPU streaming processors are available, the scheduler could initiate the execution of additional work requests. This can lead to significant performance improvements in hybrid Charm++ applications. In a message-driven model like Charm++, objects execute fuctions in response to messages received from other objects. If a function which sends a large number of messages is held up from executing, then the functions which execute in response to messages sent by this function are effectively held up as well. Suppose, for example, that we have two GPU work requests in the queue, one heavy work request which will take significant time to execute, and a second much shorter one which lies on the longest control path in the application. The standard implementation of Hybrid API will execute these work requests in separate kernels, significantly delaying the shorter, more important work request. With the asynchronous approach, on the other hand, both work requests would execute simultaneously, ensuring progress on the limiting path in the application.

This model of execution can also reduce overhead due to kernel startup. If work requests are combined to execute in the same kernel, then kernel startup overhead is incurred just once. In fact, we will present a method which could allow executing all work requests in the whole application within the same kernel.

5.2 Interface

In addition to the attributes in Hybrid API, work requests in the Asynchronous API require a parameter to indicate the number of SMs which the work request will occupy. This decision is left to the user. The choice depends on the number of blocks used by the work request and the relative importance of the work request in the application. As an example, we may want to dedicate more execution units for work requests which lie on the longest control path in the application.

5.3 Implementation

In this section we describe the implementation of Asynchronous API. While deficiencies in the CUDA programming model do not currently allow this approach to work, we believe it is instructive to present this technique of general purpose GPU programming which may become feasible with new hardware and programming models.

The key feature of Asynchronous API is dynamic assignment of work requests to the execution units on the GPU. Since kernels are serialized in CUDA, this has to be accomplished by using a single kernel executing for the duration of the application. In early versions of CUDA, it was not possible to communicate between the CPU and GPU during the execution of a kernel, which would make such an approach infeasible. More recently, CUDA allows asynchronous data transfers between the CPU and GPU while a kernel is executing. This feature was added to allow overlapping the execution of one kernel with the data transfer for a subsequent kernel. Our plan was to use this feature for a different purpose, as a communication mechanism between the CPU and the executing kernel on the GPU.

In Asynchronous API, the sole kernel is spawned by the system during the initiation of the runtime system. The kernel consists of at least as many blocks as there are SMs on the device. Having more blocks than SMs is possible, but it also presents some difficulties which we will discuss later.

A small portion of the global GPU memory in this approach is reserved for GPU/CPU control communication. Here, the main data structure is an array with one value per block of threads, via which the CPU communicates the work requests to the GPU. Initially, all values in the array are set to indicate no work has been assigned. Threads in the kernel spin on the array location for their block, waiting for the value to change, which indicates a work request assignment by the CPU.

When the value changes, the threads execute the appropriate kernel by evaluating a switch on the value in the array. This is similar to the mechanism in Hybrid API for the execution of a kernel based on the id stored in the work request, but here the control switch is performed inside the kernel on the GPU. Once the threads finish execution, one of the threads marks the work request as finished in the control array, and the threads in the block once again spin on that location, waiting for a new work request. The CPU system code will periodically read the status of the control array in GPU global memory, keeping track of which blocks are executing and which have finished. As in Hybrid API, the runtime system transfers data arrays to and from the GPU.

5.3.1 Synchronization

The above description, in the interest of clarity, avoided details regarding synchronization operations required to make the scheme work. Let us now take another look at the process, focusing on the steps where synchronization is required. (1) Before a work request executes, the system transfers the data required for execution to GPU memory. The CPU can perform this step during kernel execution only via an asynchronous memory transfer call. This operation must be synchronized to ensure the data transfer is complete by the time the work request executes. (2) Next, the system assigns the work request to a block of threads by writing an entry in the control array on the GPU. Since the array entries are written one at a time, and blocks are assigned no more than a single work request to execute, this step does not require synchronization. (3) By periodically monitoring the control array, the system can recognize when a work request is finished. (4) Before data can be transferred back to CPU memory, we need a way to ensure that GPU memory writes performed by the work request have committed. (5) The system copies data to CPU memory via asynchronous memory transfer. As in step (1), this needs to

be synchronized.

As we can see, steps (1), (4), and (5) require synchronization. The synchronization after CPU-GPU data transfers should technically be possible through the *stream* feature in CUDA. Note that a stream in this context is not the same as the streams discussed in Chapter 2. The CUDA stream construct exists as a way to introduce concurrency of memory transfers and kernel execution into the language. CUDA streams are integer inputs to kernels and memory transfers, and thus exist only in the context of the CPU section of a CUDA program. The semantics is that within a stream, operations execute sequentially, but that different streams can execute concurrently when the model allows it. A synchronization call performed on a stream should synchronize just the operations within that stream, and have no effect on other streams. We tried using this feature to implement memory transfer synchronization by running the transfers for each block in a different stream. We found that in the current implementation of CUDA, synchronizing on a stream waits not just for operations in that stream to commit, but also waits for any currently executing kernel, regardless of the strem. As a result of this behavior, we were unable to implement Asynchronous API in CUDA.

It should be noted that the synchronization required in step (4) is of a different kind than the data transfer synchronizations just described, as it involves synchronizing memory transfers from the GPU to the CPU with memory writes performed locally on the GPU. While CUDA provides a barrier function to synchronize all threads in a block, it is unclear whether this has the effect of waiting for all memory writes to commit.

Overall, our approach broke some generally accepted rules about CUDA programming by trying to communicate between the CPU and the GPU during kernel execution. While it may thus not be surprising that it was not possible to implement Asynchronous API in CUDA, we believe our work shows how a more generalized

version of CUDA could allow for new applications and a shift in granularity considerations on the GPU.

5.3.2 Grid Structure

One of the aspects of the implementation which has been glossed over so far is the structure of the grid for the kernel executing on the GPU. In the simplest approach we could have the number of blocks equal the number of SMs on the GPU. This should in practice ensure that each block gets assigned to a different SM. Sending a work request to a block would then be equivalent to assigning work to one SM. The CUDA programming model encourages having more blocks than there are SMs in order to increase the number of threads per SM and allow for greater potential for overlap between computation and memory access. It would be tricky to make this work for Asynchronous API, however. If we have more blocks than there are SMs, CUDA gives no guarantees about how the blocks will be assigned to the SMs. This could lead to imprecise behavior, as the system would be unable to tell exactly how many SMs will be used for a particular work request. In practice, having blocks with a large number of threads (i.e. 256 or 512) should provide for a high enough degree of overlap between computation and memory access when we only have a single block per SM.

5.3.3 Load Balancing

different block sizes

5.4 Asynchronous Memory Transfer

5.5 Power Considerations

It should be noted that our approach keeps the GPU busy throughout the duration of the execution for the application by busy waiting with the threads which are not executing. When executing, GPUs consume more power than in idle mode.