GPU Programming in a High Level Language

Compiling X10 to CUDA

Dave Cunningham Rajesh Bordawekar Vijay Saraswat

IBM TJ Watson

{dcunnin,bordaw,vsaraswa}@watson.ibm.com

Abstract

GPU architectures have emerged as a viable way of considerably improving performance for appropriate applications. Program fragments (kernels) appropriate for GPU execution can be implemented in CUDA or OpenCL and glued into an application via an API.

While there is plenty of evidence of performance improvements using this approach, there are many issues with productivity. Programmers must understand an additional programming model and API to program the accelerator; concurrency and synchronization in this programming model is typically expressed differently from the programming model for the host. On top of this, the languages used to write kernels are very low level and thus prone to the kinds of errors that one does not encounter in higher level languages. Programmers must explicitly deal with moving data back-and-forth between the host and the accelerator. These problems are compounded when the user code must be run across a cluster of accelerated nodes. Now the host programming model must further be extended with constructs to deal with scale-out and remote accelerators. We believe there is a critical need for a single source programming model that can be used to write clean, efficient code for heterogeneous, multi-core and scale-out architectures.

The APGAS programming model has been developed for such architectures over the past six years. APGAS is based on four fundamental (and architecture-independent) notions: locality, asynchrony, conditional atomicity and order. X10 is an instantiation of the APGAS programming model on top of a base sequential language with Java-style productivity. Earlier work has shown that X10 can be used to write clean and efficient code for homogeneous multi-cores, SMPs, Cell-accelerated nodes, and clusters of such nodes. In this paper we show how X10 programmers can write code that can be compiled and run on GPUs. GPU programming idioms such as threads, blocks, barriers, constant memory, local registers, shared memory variables, etc. can be directly expressed in X10, and do not require new language extensions. We present the design of an extension of the X10-to-C++ compiler which recognizes such idioms and produces CUDA kernel code. We show several benchmarks written in this style. The performance of these kernels is within 80% of hand-written CUDA kernels.

We believe these results establish X10 as a single-source programming language in which clean, efficient programs can be written for GPU-accelerated clusters.

Categories and Subject Descriptors D [*3*]: 3

General Terms Languages, Performance, Human Factors

Keywords CUDA, GPU, X10, Heterogeneous, Distributed, Multicore, Parallel, High-level, Java.

1. Introduction

Clusters of GPU-accelerated nodes are becoming increasingly important for high performance and scale-out computing. For instance the Tianhe-1, organized as a cluster of NVidia GPU-accelerated x86 nodes leads the Top500 list of super computers, as of November 2010. GPUs can often provide over 30x acceleration over CPUs for dense numeric kernels.

Currently such machines are programmed using an ad hoc collection of different frameworks. C or Fortran is used as a base sequential language, with MPI for scale-out. GPU kernels are writtne in CUDA or OpenCL. Programming in such an environment is complicated and presents a barrier to entry for those who do not have a background in HPC and systems programming.

We believe that there is an urgent need for a language with Java-style productivity that can be used to program clusters of GPU-accelerated nodes. We believe such a model should satisfy the following requirements (in order of priority):

- A single source program should be capable of specifying execution over a host, an attached GPU, and across a cluster.
- The model should have a uniform set of constructs that are used to deal with concurrency within a node and across nodes.
- Performance should be consistent with conventional models.
- The model should be high-level, hiding details of data transfer between nodes as far as possible.

1.1 The APGAS Model

The APGAS model [11] has been developed in the last few years to address programming clusters of (possibly Cell-accelerated) multicore nodes. It is organized around four basic and orthogonal concepts: *locality*, *asynchrony*, *conditional atomicity* and *order*.

Places Computation is organized around one or more places. A place contains (mutable) state as well as activities that operate on that state. Places are not required to be homogeneous: in a single computation, one place may be mapped to an x86 core, or an x86 node (with multiple cores corresponding to it), or an SMP, or a GPU.

Places are reified. They can be stored in variables, passed into functions, etc, as an instance of the struct `x10.lang.Place`.

Let us call a unit of serial execution an *activity*. Activities are located in a single place for their lifetime. Given a place `p` the statement `at(p) S` can be used to request execution of the statement `S` at place `p`. The execution is synchronous: control for the current activity transfers from the current place (say, `q`) to place `p`, executes `S`, and then returns to place `p` to execute the next statement after `S`.

More details on place discovery, naming and properties are given below.

Asynchrony An activity may use the statement `async S` to launch a new activity to execute the statement `S`. `S` may reference variables in the surrounding lexical environment.

Conditional Atomicity An activity may use the statement `when (c) S`, where `c` is a `Boolean`-valued expression called the *condition* and `S` is a statement. The execution of this statement in a state s terminates in a single step and yields the state s' if and only if the condition `c` is true in `s`, and execution of `S` in `s` yields `s'`. Note that no other activity is permitted to execute when the state is transitioned from `s` to `s'`, i.e. the `when` statement is *atomic*.

In the following we will allow ourselves to use `atomic S` to mean `when (true) S`.

Order There must be a mechanism for imposing a partial order on the state changes caused by individual activities. The statement `finish S` executes `S` and then waits for all activities spawned during the execution of `S` to terminate (recursively). Thus it ensures that all activities spawned after the `finish` see the effect of the state changes by the activities spawned before or during the execution of `finish`.

The APGAS model exploits lexical scoping. Variables declared in an outer scope can be accessed within the body `S` of an `async S` or `finish S` or `at (P) S` statement. In the case of `at (P) S`, the variables accessed in `S` that are declared in an outer scope are subject to an *at-transfer*. That is, the values of these variables are serialized, transported to `P` and deserialized into shadow variables with the same name. Thus the code `S` always accesses a local copy of "outer" variables. If the variables are pointers to heap objects, the copy is deep, i.e., a fraction of the object graph is transferred and recreated on the target place.

The `async`, `finish` and `at` statements can be nested arbitrarily. This is a fundamental part of the flexibility of the APGAS programming model. For instance, `async at (P) S` is the typical pattern used to express an *active message* [16] to `P`. The message asks for the code `S` to be executed at `P` asynchronously (with outer variables referenced in `S` implemented through at-transfer). Recursive parallelism (a la Cilk) is expressed with the idiom `finish { ... async S ...}` , i.e. with an outer `finish` enclosing a number of asyncs. Static partitioning of work across `P` workers can be done by using the idiom `finish for ([i] in 1..P) async S`, where `S` is sequential code expressing the execution of the `i`th chunk of work.

Finally, X10 defines a notion of *clocks* using `when`. A clock is a barrier-like construct. An activity may be registered on multiple clocks $c_1, \ldots c_N$. It may spawn other activities and specify that they be registered on one or more of $c_1, \ldots c_N$. An activity uses the construct *next* to signal that it has finished the work associated with the current phase of the clock and to wait for the clock to advance. The clock advances only when all activities registered on the clock have executed `next`.

1.2 The GPU Programming Model

GPUs are made available to host applications via the CUDA[10] and OpenCL[7] frameworks. Here, we abstract from CUDA and OpenCL and consider a generalized GPU programming model. However since the terminology between the two implementations is different, and our implementation is based on CUDA, we choose to use the CUDA terminology in this paper.

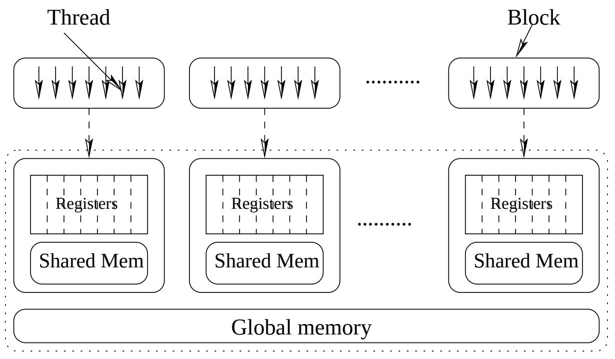


Figure 1. The components of the CUDA programming model

The GPU programming model provides programming language constructs that expose the peculiarities of the GPU architecture so that programmers can write efficient programs. There is a distinction made between memory that exists on the GPU and on the host. Only GPU code can access GPU memory and likewise for the host. However the host can allocate and free GPU memory, and invoke asynchronous copies between its own memory and the GPU memory.

The only code that can run on the GPU is a kernel. A kernel is a block of code in a restricted language that is executed by hundreds of *threads* simultaneously. Competitive performance is only possible when the GPU is running very large numbers of threads.

The memory on the GPU consists of the *global memory*, which is the memory that is managed by the host. However this memory is slow for each thread to access, so there are also other kinds of memory that are available while the kernel is running. Each thread has its own set of registers, which are exposed as scoped variables in the programming model. For communication between threads, there is *shared memory*[1]. The shared memory is arranged such that the threads are partitioned into *blocks* and each block has its own shared memory, as shown in fig. 1. The host program dynamically decides how many blocks, how many threads per block, and how much shared memory per block should be used, when invoking a given kernel. To synchronize accesses to shared memory, there is one construct provided, an SPMD barrier represented by the function `__syncthreads()`, which causes a thread to wait until all threads in the block has reached the barrier.

Finally, there are some special ways of accessing global memory. Constant memory is a small amount of memory that can be used by all threads, and is almost as efficient as local registers. It is read-only and cached. Texture memory is another form of read-only, cached memory, except that the cache is very much larger and the latencies are higher. Texture memory also provides some unusual indexing operations, where indexes are given in floating point and array elements are filtered (interpolated). The configuration of constant and texture memory are exposed through additional API functions that must be called on the host before the kernel is invoked.

[1] Since shared memory is a commonly used term in concurrent programming, to avoid confusion we do not use it except when referring to CUDA shared memory.

1.2.1 Preview of GPU programming in APGAS

We briefly preview how GPU concepts are represented in APGAS, details are provided in section. 2. A GPU is reified as a place, say `P`. The global memory of the GPU is the heap at `P`. Shared memory mapped to local variables declared within the body `S` of an `at(P) S`. GPU threads are represented using `async`, blocks using for loops, with completion detected using `finish`. Barriers are implemented using clocks. Registers and local memory are represented through local variables in the async.

1.3 Our Claims

- The APGAS programming model in general, and X10 in particular, can be used to program GPUs productively.
- GPU programming elements have a direct expression using X10 constructs
- We have extended the X10 to C++ compiler to understand the X10 presentation of CUDA idioms and generate CUDA code from it.
- The resulting code executes within 80% of the performance of native CUDA kernels for given benchmarks.

Rest of this paper. We describe how to express the GPU programming model within the APGAS programming model in section 2. In section 3, we explain how we modified the X10 compiler and runtime to target CUDA, and we give our experimental results with these modifications in section 4. We discuss related work in section 5 and conclude with section 6.

2. Expressing GPU Programming in APGAS

We now show how we use APGAS constructs to express GPU programming idioms. This allows X10 programmers to write programs that use the GPU. In the compiler and runtime, we recognise when programs are using the APGAS constructs in this fashion, and arrange for the code to be executed on the GPU instead of the CPU.

2.1 Places and Global Memory

Given the hierarchal nature of the memory on a GPU, it is not obvious how to define a place. We could have broken the GPU up into 1 place per block. However we decided to treat the whole GPU as a single place. The reasoning behind this decision was that the heap at a place should be filled with objects that live for a long time. The only memory in the GPU whose allocation outlives the execution of the kernel is the global memory (and the memories that are simply cached views on global memory). Local memory does not outlive the thread in which it is defined, and shared memory does not outlive the block that accesses it. Since global memory is amorphous, with no internal divisions or affinities, it made sense to group it together as a single place, with its heap represented by objects in global memory. Therefore to run code on the GPU we use the `at` construct.

Consequently, the set of places in an executing program forms a two level tree. At the top of the tree are the host places, equally able to communicate with any other host place, thus forming a totally interconnected nest. Under each host place there can be an arbitrary number of accelerators. Communication between a host place and its accelerators will clearly be faster than communication between a host place and some other host's accelerator, so we chose to expose this relationship. In an APGAS language, one can provide an API for iterating over hosts, accelerators of a host, and for finding out whether a given place is a host, a CUDA GPU, or perhaps some other kind of accelerator. For programmers, these functions are analogous to the functions in the CUDA/OpenCL API for discovering the GPUs on the local system. The X10 code in fig. 2 executes a kernel on each CUDA GPU on the local host.

```
for (accel in here.children().values()) {
    if (accel.isCUDA()) {
        at (accel) {
            ...
}   }   }
```

Figure 2. Using the `at` construct to invoke code on a GPU.

2.2 Allocating Memory on the GPU

Memory allocation in Java-like languages is expressed with the `new` construct. In a Java-like APGAS language like X10, remote allocation is simply expressed by doing a `new` inside an `at`. However the GPU programming model does not allow memory allocation inside a kernel[2]. Instead, GPU programmers call an API on the host in order to allocate memory. We had to choose between recognising certain combinations of `at` and `new`, or exposing a library call that took a place and allocated an object there. We chose the latter because it is easier to build a more robust implementation. If the place is a GPU place, the call is implemented with CUDA operations. Otherwise, it is implemented with `at` and `new`. The following example demonstrates ways in which we can make an array on the GPU place `p`, from the host place, where `T` is the type of the elements of the array (can also be inferred from the arguments).

```
CUDAUtilities.makeRemoteArray[T](p, sz, single_value);
CUDAUtilities.makeRemoteArray[T](p, sz, (i:Int)=>expr);
CUDAUtilities.makeRemoteArray[T](p, clone_this_array);
```

In future we will investigate using the kernel `malloc` function in order to supprt `new` in kernels and thus make allocating memory on the GPU more natural. This will also allow creating objects other than arrays.

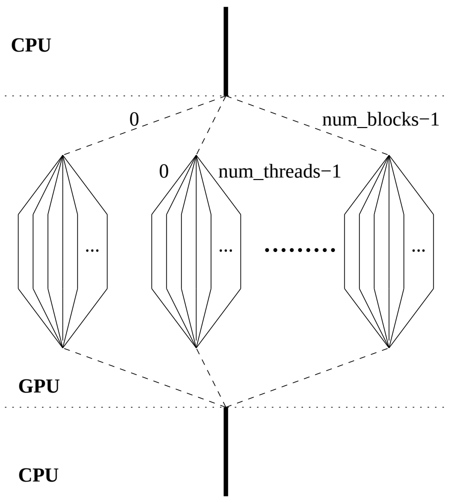
2.3 Copying Memory to/from the GPU

When programming kernels, data that needs to be transferred each kernel invocation can simply be captured from the enclosing scope, instead of copied explicitly. One common operation we would like to optimize is the updating of an existing array on the GPU. In fact this is generally useful for APGAS languages, and in X10 we have provided an API that mirrors Java's `System.arrayCopy`, except it allows one of the arrays to be a remote reference, and is asynchronous, notifying the governing `finish` statement upon termination like an `async` statement would. In the following example, `r` and `l` are remote and local arrays, respectively. The other parameters are integers. The function is overloaded to allow copies in the opposite direction by swapping the `r` and `l` parameters.

```
finish Array.asyncCopy(r, r_off, l, l_off, len);
```

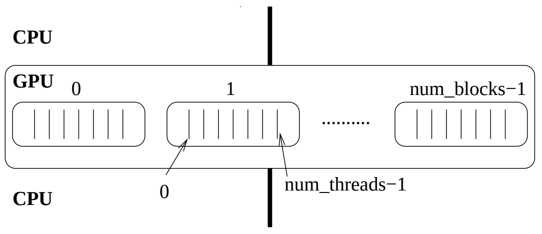
This API can be implemented with the basic APGAS operations, by capturing the source data, switching to the remote place and using a loop to fill in the remote array. However to do so would mean creating objects and incurring serialization overhead. To this end we provide a native implementation of these copy functions that is more efficient. In the case of copying to a GPU, we can implement these same copy functions on top of the CUDA API's memory copy operations. Thus, copying data between a local host array and a GPU array can be achieved through exactly the same code as a programmer would write for a normal distributed program that did not use GPUs.

[2] In fact at the time of writing this has just become possible in CUDA but it was too late to influence our design



```
at (p) {
    finish for ([b] in 0..num_blocks-1) async {
        val shm = new Array[Int](num_elements, init);
        finish for ([t] in 0..num_threads-1) async {
            ...
}   }   }
```

Figure 3. Specifying blocks, threads and shared memory in X10.



```
__global__ void kernel (int *gpu_init)
{
    __shared__ int shm[NUM_ELEMENTS];
    // initialize shm from gpu_init
    ...
}
...
// allocate gpu_init on gpu
// initalize gpu_init via memcopy
kernel<<<num_blocks,num_threads>>>(gpu_init)
```

Figure 4. Specifying blocks, threads and shared memory in CUDA.

2.4 Threads, Blocks, and Shared Memory

In the GPU programming model, threads are arranged into blocks, where threads within a block may communicate via shared memory. In APGAS the equivalent of a thread is an activity, which we create with the `async` construct. To create many activities, one uses a loop to execute multiple `async` constructs. In fig. 3, we run with `num_blocks` blocks each with `num_threads` threads. The syntax allows for a multidimensional iteration, which is specified by using, e.g., `[bx,by,bz]` instead of `[b]`, however we only present 1-dimensional iterations in this discussion. For comparison, we give the equivalent CUDA code in fig. 4.

In the X10 code, to represent that kernel termination is dependent on the termination of the blocks, we use the `finish` construct around the outer loop. Likewise, we use another finish to represent that the block terminates when all of the threads terminate. The number of activities is actually $\texttt{num\_blocks} * \texttt{num\_threads} + \texttt{num\_blocks}$, although this maps down to only $\texttt{num\_blocks} * \texttt{num\_threads}$ CUDA threads. This is because the other activities are only used to represent the parallelism between blocks, and are removed by the compiler. In fig. 3 they are represented by dashed lines.

Shared memory objects are objects on the heap, but are not permitted to live beyond the lifetime of a block. This can be expressed by allocating objects between the two loops. We have to enforce the restriction that the objects may not leak from the block, because this would allow dangling pointers after the shared memory becomes inaccessible. This restriction can be implemented e.g., by disallowing access to the array pointer itself, only allowing indexing operations. The code in fig. 3 shows how a shared memory array can be allocated in X10.

```
val tmp1 = num_blocks-1;
val tmp2 = num_elements;
val tmp3 = num_threads-1;
at (p) {
    finish for ([b] in 0..tmp1) async {
        val shm = new Array[Int](tmp2, init);
        finish for ([t] in 0..tmp3) async {
            ...
}   }   }
```

Figure 5. The compiler executes some subexpressions of the kernel on the host.

We must restrict the subexpressions within this pattern, in order to conform to the GPU programming model. The only part of the pattern that is actually run on the GPU is the body of the innermost async, and the initialization of the shared memory. The number of blocks and threads and the amount of memory must be known before the kernel is invoked. We require the expressions `num_blocks`, `num_threads`, and `num_elements` to be hoistable from within the at statement and executable on the host, as shown in fig. 5.

For hoisting to be safe, these expressions must be highly restricted. They must be *place-independent* in order to have the same meaning on any place. This means no use of the construct `here`, which evaluates differently at each place, and no field accesses, which can only be performed at the place where the objects live. They also need to evaluate to the same value each loop iteration. One simple rule that suffices for our purposes, but could be greatly generalized, is to restrict the expressions to only allow local variables defined outside the at expression, and a handful of operations such as integer arithmetic, that are known to be safe when hoisted in this manner. This is enough for the benchmarks we discuss later. Even if it were not, one can take hoistable code that is conservatively rejected, and hoist it manually, resulting in a program that passes checking and has the same behavior and performance.

2.5 Barrier

Shared memory can be used in a read-only fashion, as a cache. However it is much more useful as a staging ground into which to stream from global memory, and in this context it needs to be mutable. Mutable shared memory requires synchronization, and GPUs provide this with a call `__syncthreads()` that behaves like a barrier within the block. The analogous construct in APGAS is the clock. Clocks are more general, since they allow participants to join and leave the clock at will. However we can restrict the code to prevent this, and then the semantics align with that of the GPU

```
at (p) {
    finish for ([b] in 0..num_blocks-1) async {
        clocked finish for ([t] in 0..num_threads-1)
        clocked async {
            ...
            next;
            ...
}   }   }
```

Figure 6. Using clocks on the GPU to represent the barrier construct

```
at (p) {
    local cmem = arr.sequence();
    finish for ([b] in 0..num_blocks-1) async {
        finish for ([t] in 0..num_threads-1) async {
            ...
}   }   }
```

Figure 7. A simple way to express many uses of constant memory.

programming model. In order to use clocks in a kernel, the X10 code is shown in fig. 6.

Each block has its own clock, indicated by the `clocked` keyword on the `finish` (we do not mark the outer `finish` as `clocked` since there is no global barrier in the GPU programming model). The `async`s that form the blocks are also marked clocked, and this allows the use of `next` within them. The scope of the clock is limited to the `finish`, so only the threads within the block executing the `next` will be synchronized.

2.6 Kernel Parameters, Registers and Local Memory

GPU kernel parameters are represented in APGAS as the capturing of local variables within the `at` construct. In the case of pointers, APGAS goes further and creates an object graph on the GPU. However, for variables of primitive type (and in X10, struct type) that involve no indirection, the semantics of kernel parameters and the capture of scoped variables coincide.

Registers and local memory are what would normally be called a stack. They are local to each thread, and can be considered part of the sequential subset of GPU programming. APGAS extends the sequential subset of whatever language in which it is used, so in X10 we represent registers and memory in the same way as CUDA and OpenCL, through local variables on the stack.

2.7 Constant Memory

At present we expose one specific use of constant memory, as a cache that is updated before each kernel. We expose this in a similar way to shared memory except for two differences. Firstly, it is read-only, and in X10 a read-only array is expressed with a `Sequence` object that is an immutable view on a mutable array. Secondly, all blocks share the same constant memory, so we place the definition within the kernel but outside the outer loop as shown in fig. 7.

Since `cmem` will be refreshed on any subsequent kernels, references to it may not outlive the current kernel, which is a restriction we also apply to shared memory objects. We can easily enforce this by, again, only allowing indexing operations on the array.

The GPU programming model allows constant memory to be initialized once and then used in many kernels. However, we cannot represent this just by allocating memory on the GPU with `new`, since objects in constant memory are statically defined in the GPU programming model. In future we could look at using arrays in static fields to represent this more general idiom, thus avoiding the performance cost of reinitialising constant memory on every kernel invocation.

2.8 Texture Memory

Texture memory can be expressed as a datastructure. One needs a way to create a texture from the host on a given GPU, giving the dimensions, the content, and the kind of filtering that is desired. A pointer to the texture object would then be captured by the kernel, and a method called on it to do the texture fetches at the given coordinates. In order to support the same code running on the host, the semantics of the texture fetches, including the filtering operations, would have to be implemented on the CPU. This would obviously not be as fast as the dedicated hardware on the GPU.

2.9 Restrictions On Kernel Code

Earlier, we described the restrictions on the loops and shared memory definitions that are due to the necessity of hoisting sub expressions out onto the host instead of executing them on the GPU place as specified. There are also other restrictions that are more fundamental to the GPU that govern the *actual kernel*, the body of the innermost `async` block.

Until very recently in CUDA, kernels were not permitted to allocate memory. We thus do not allow `new` inside the actual kernel. There was also no indirect branching supported by CUDA GPUs until the latest revision of the architecture, so all function calls had to be final or static. These limitations are still present in older versions of the hardware/software and are still present in OpenCL. Thus our restrictions of the language are still relevant even though the trend is for greater expressiveness on the GPU.

The actual kernel must be sequential non-distributed code, i.e., no occurences of `async`, `at` or `finish`. Recursion is not supported on GPUs, so we also must not allow it. Recursion is easy to detect in the context of static binding or whole program knowledge.

There are also limits on the number of threads, number of blocks, and limits on the amount of shared memory that can be created. If these limits are broken, we can simply throw an exception at runtime.

3. Implementation

To run GPU programs written in the APGAS model, we added a CUDA code generation backend to the X10 compiler, and extended the X10 runtime environment to be able to invoke the generated kernels. We chose CUDA instead of OpenCL because at the time OpenCL was not widely supported. However everything we have achieved could also now be achieved with OpenCL.

For simplicity we have restricted ourselves to 1-dimensional iterations over blocks and threads. This makes some kernels more complicated as they have to compute the x and y coordinate explicitly from the single variable. We also require the clocked form of the inner loop, seen in fig. 6, even if `next` is not used. This means we can trivially compile `next` into `__syncthreads()`.

We currently only support int and float types in the kernel, and the only objects that can be created on the GPU's heap, shared memory, or constant memory, are arrays. These are severe restrictions but kernels do not typically use complicated object graphs or do much pointer chasing. Instead they stream arrays and work with primitives in registers and shared memory. We also have not implemented support for texture memory.

The X10 compiler does not know whether a place `p` is a GPU place or a host place, since this depends on how the program is launched. A kernel that begins with `at (p)` may run on a CPU if `p` happens to be a host place. Thus, we must compile kernels to CUDA and CPU code, dispatching at runtime.

Given the strict rules governing a kernel, there is the possibility of an `at` with a non-conforming body being targeted at a GPU place. We cannot enforce the restrictions on all `at` blocks, since this would affect host code, and we do not know which will run on the GPU. Thus, we must raise an error at runtime if the code is not suitable. Because this can cause errors to remain undetected, we decided to require an `@CUDA` annotation on `at` blocks that are intended to be CUDA-capable. This way, if the programmer accidentally breaks one of the rules they will get a static error. Any block that is executed on the GPU without a `@CUDA` annotation yields a runtime error.

3.1 Compiler

Our modified X10 compiler takes a `@CUDA`-annotated `at` in class C, pattern-matches the constant memory, loops, array definitions, etc., and compiles the body of the innermost async into a C.cu file (a kernel written in NVIDIA's CUDA language which is a extended subset of C++). Each kernel is named using an ascending number, and is marked with `extern "C"` to avoid C++ name mangling. Since there is already a C++ backend for X10, we were able to reuse a lot of code generation for the basic sequential constructs, like conditionals, loops, variable declarations, etc. It was trivial to map `Math.sqrt` and other such functions to their CUDA equivalents. The compiler then invokes nvcc to compile the cu file to a cubin file (object code for the GPU).

On the host, an X10 array is represented by an instance of the `Array[T]` class. This encapsulates a pointer to a backing store, so there is one more level of indirection than is necessary. In addition to this, the `Array` class implements a very high level notion of non-zero-based multidimensional array. Worried about the performance implications of this, we restricted the usage of `Array[T]` to simply allow 1-dimensional indexing and retrieving the size of the array. We then represented it with a struct containing only the pointer to the backing storage and the size of the array. We use this for shared memory and constant memory arrays as well as global memory arrays. If the length is not used by the program, it is optimized away by nvcc, i.e., it does not consume a register.

We also generated code for the host to provide data needed to invoke the kernel. This code is placed into a *pre* callback (described in detail in section. 3.2.1). In this callback, the variables captured by the kernel are arranged into a struct called `env` and passed to the kernel with a single parameter. This ensures the alignment is correct in the kernel argument memory, whereas individual parameters would need to be packed carefully.

Using a struct also allows for an alternative when `env` is bigger than the number of bytes available for kernel parameters. In this case, a global memory buffer is allocated on the GPU by the host in the `pre` callback. The allocated buffer is large enough to hold `env`, and the arguments are copied to it before the kernel is invoked. The kernel has a single parameter, a pointer to `env`, and the arguments within are copied into shared memory variables by the 0th thread in each block before the user code executes. This latter approach is currently the default, since the number of captured variables is unlimited. If the number of arguments is small enough, the former behavior can be specified with the additional annotation `@CUDADirectParams` next to the `@CUDA` annotation. We leave automatically making this choice as future work.

We implement all shared memory using CUDA's dynamic shared memory support, where the amount of shared memory required is given at kernel invocation time instead of statically. The size of each shared memory array is calculated on the host by generating C++ code from the first argument in the `Array` constructor in each shared memory definition. The total size is given to CUDA when invoking the kernel. In the kernel itself, the shared memory is initialized by the 0th thread in each block, and a barrier is used to hold back the other threads until initialization is complete. Constant memory is initialized by the host before the kernel is invoked, which we generate code for in the `pre` callback.

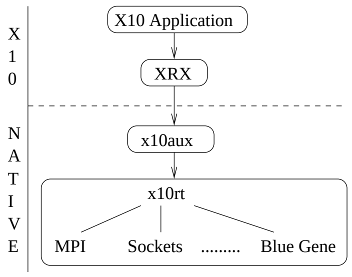


Figure 8. The original software modules in the X10 runtime

Finally, we found that constant propagation was invaluable for improving the performance of our test kernels. In a Java-like language, i.e., without a mechanism like C's #define, one relies on final variables to hold constants. With constant propagation, constants defined in static fields and in final variables in scope will be inlined into the kernel. This is beneficial firstly because it reduces the amount of data needing to be sent to the kernel at invocation time, and secondly because these values no-longer need to be held in registers, decreasing the overall register pressure.

3.2 Runtime

The X10 runtime is organised in several layers, as shown in fig. 8. At the highest level the APGAS constructs are desugared (using a closure) into calls into a private API, *XRX*, which is implemented in X10 and handles finish states, termination propagation, intra-place load balancing, and more. XRX then passes control to a C++ API, *x10aux* which asynchronously invokes a closure at the target place with a finish state as an argument. The implementation of x10aux serializes the closure and finish state into a buffer and then uses a C API *x10rt* to move the data to the far side. The reason for this design is that x10rt and XRX can be reused with the X10 Java code generation backend whereas x10aux is specific to the C++ backend.

The x10rt library is responsible for finding out the number and configuration of places, sending messages to a given place, and keeping a register of callbacks from the X10 program for newly received messages. We needed to make major changes to `x10rt` to support invocation on GPUs, and minor changes to x10aux. We wanted to have a tree of places, with zero or more accelerator places under each host. We accomplished this by renaming the existing API *x10rt_net*, and hiding it beneath a new API, *x10rt_logical*. We wanted to reuse the existing networking code, which is implemented for a variety of backends. Also underneath `x10rt_logical` is a new API called *x10rt_cuda*. The `x10rt_logical` layer therefore provides its own notion of places, a coherent view on the two independent layers underneath it, see fig. 9.

3.2.1 Initialization

While CUDA is being initialised at each host place, the hosts are also communicating and initializing their network. After both are complete, `x10rt_logical` takes information from both and communicates with other hosts to build a complete tree of uniquely named places in the system, consistent across all hosts in the system. We defined functions for inspecting the tree in the `x10rt_logical` API, which is now the public face of `x10rt`. This complements the functions for sending messages to places and registering callbacks for receiving messages at the local place, whose signatures and names are taken from the old API `x10rt_net`. We then extended the layers above `x10aux` and `XRX` to expose the structure of the tree to the X10 programmer via the `x10.lang.Place` class.

We had to extend the callback registration API to accommodate CUDA execution, because there is work to do on the host before a kernel can be invoked, and this work depends on the kind of

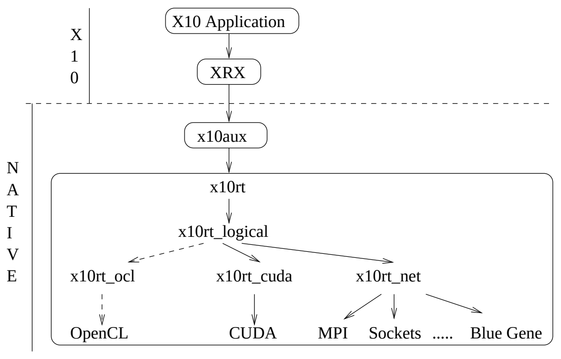


Figure 9. Our revised software modules in the X10 runtime. The dashed line represents potential future expansion.

code in the kernel (such as setting the kernel parameters, refreshing constant memory, and uploading objects to global memory). There is also code that must run on the host after the kernel is completed to implement `finish`. Therefore instead of a host registering a single callback to handle a message, it additionally supplies the names of the cubin file and the kernel, so that `x10rt_cuda` can load the code, and also a pair of callbacks, called *pre* and *post*, that will run on the host before and after the kernel runs on the GPU.

During the program's initialisation time, the X10 runtime opens each cubin file, and loads the appropriate kernels. The `pre` and `post` callbacks allow the user of the X10RT API to do arbitrary work before and after their kernel has executed. In `x10aux` the `post` callback for every kernel just updates the finish state via a call to `XRX`. The `pre` callback is generated for each kernel as described previously.

3.2.2 Routing

It is possible in APGAS to launch an `at` block at any place in the tree, including GPUs that are not local to the host in question. The design of `x10rt_logical` is intended to allow transparent routing of these messages to the GPU's host. We have not yet implemented routing but believe it is not hard.

3.2.3 Implementing `at`

In the APGAS model, the fundamental constructs for invoking an asynchronous message are `async` and `at`. However in the implementation, for performance reasons, an asynchronous message is the fundamental operation, and `at` is implemented by using another message to notify termination. The constructs `async at(p)` are optimized to not create an activity at the source. We did not implement kernel invocation for an `at` by itself but instead trapped the more fundamental `async at`. To wait for a kernel to complete one must do `finish async at` instead of just `at`.

To all layers except `x10rt`, the `async at` implementation remains almost the same. The object graph captured by the kernel is serialized as usual, and given to `x10rt`. The `x10rt_logical` layer then dispatches the buffer to `x10rt_cuda`, which calls the `pre` callback. The `pre` callback must provide the number of blocks / threads, the amount of shared memory, the contents of the kernel argument buffer, and the content of the constant memory.

When the callback returns, `x10rt` can invoke the kernel. We used the CUDA driver API since we are packing our own kernel parameters and we are only generating CUDA code to implement the kernels. We use a single CUDA stream for kernel invocations, to separate them from DMAs which occur in parallel. While the stream is busy, the kernel is queued in `x10rt`. Either way, control immediately returns to the application. The `x10rt` library was designed such that all calls are asynchronous and the application must call `x10rt_probe` regularly, e.g., when waiting at the end of a `finish` block, to cause progress within `x10rt`. Within this call, the network buffer was checked for new messages and callbacks dispatched. We extended the call to also monitor the CUDA streams, and when a kernel terminates, to call the post callback.

3.2.4 Automatically choosing the number of blocks/threads

When writing one of our benchmark applications, we discovered that to get maximum performance, the number of blocks and threads had to be precisely chosen based on characteristics of the kernel and the particular GPU it runs on. This causes a problem when trying to write portable code that runs fast on a variety of GPUs. To solve this, we developed a heuristic to choose them automatically. The rules are as follows:

- Preferred: More blocks and fewer threads. This means fewer threads will be stalled by a barrier.
- Preferred: Maximise occupancy, i.e., maximise the total number of threads that are running concurrently. This helps mitigate memory latency.
- Required: The number of threads is a multiple of 32, this is the number of threads that the architecture will execute per instruction fetch.
- Required: The number of threads is a multiple of 64, this causes shared memory access to be more efficient.
- Required: The number of threads per block ≤ 512.
- Required: Each MP (NVIDIA terminology for what is conventionally called a core) in the GPU is equally loaded.
- Required: Each MP executes no more than the maximum number of threads that it can concurrently execute.
- Required: Few enough blocks per MP that the shared memory available at each MP is not exceeded by the needs of the blocks executed on each MP. Note that the amount of shared memory required is dynamically chosen by the application at kernel invocation time.
- Required: Few enough blocks and threads that the number of registers available at each MP is not exceeded by the number of registers required by the threads executed on that MP.

To find the optimal number of blocks/threads given these rules, just prior to kernel invocation we iterate down a list of (blocks,threads) pairs and pick the first conforming pair. The ordering of the list causes preferred pairs to be selected first. The list gives the number of blocks per MP, we then multiply this by the number of MPs in the given GPU. Below is the list of (blocks,threads) pairs we used:

```
{ (8, 128), (4, 256), (2, 512), (5, 192), (3, 320), (7, 128),
  (2, 448), (6, 128), (4, 192), (3, 256), (2, 384), (5, 128),
  (2, 320), (3, 192), (8, 64), (4, 128), (2, 256), (1, 512),
  (7, 64), (1, 448), (6, 64), (3, 128), (2, 192), (1, 384),
  (5, 64), (1, 320), (4, 64), (2, 128), (1, 256), (3, 64),
  (1, 192), (2, 64), (1, 128), (1, 64) }
```

To use this feature, the kernel must be written to work properly no matter how many blocks/threads there are. This works well for kernels that are strip mining a long array. However, some kernels divide the work into blocks and threads based on the dimensions of datastructure they are working on, so these kernels would need to be rewritten to make use of this feature. Some kernels may allow a smaller range of possibilities, or prefer a different kind of heuristic, so we may in future allow the list to be specified by the user.

The code that chooses blocks and threads is accessed via the `x10rt` API and is called from the `pre` callback. We expose it to X10 programmers as shown in fig. 10. This is an exception to the rule that the loop bounds expressions must not use variables scoped

```
at (p) @CUDA {
    val a_b = CUDAUtilities.autoBlocks();
    val a_t = CUDAUtilities.autoThreads();
    finish for ([b] in 0..a_b-1) async {
        clocked finish for ([t] in 0..a_t-1) {
            clocked async {
                ...
}   }   }   }
```

Figure 10. Automatically choosing blocks/threads for a kernel

from within the at construct. The compiler recognizes the calls and generates special code in the pre callback to call the `x10rt` function. It then generates code for the loop bound expressions in an environment where `a_t` and `a_b` are in scope and initialized to the values returned by the `x10rt` function.

When `p` happens to be a host place, the `CUDAUtilities` functions return 8 blocks and 1 thread. This is helpful because activities on the host are more heavyweight than on the GPU.

The simple heuristic to automatically choose blocks and threads is independent of the rest of the work presented here, and could be exposed as a utility library and used in any CUDA application.

3.2.5 Implementing Remote Array Allocation and Copies

We had to add API calls to `x10rt` for the allocation and deallocation of memory. These simply map down to the underlying CUDA API calls that do the same. In XRX we currently only support allocating arrays on the GPU. As discussed in section. 2.2, we expose this to the programmer this through the following API calls:

```
CUDAUtilities.makeRemoteArray[T](p, sz, single_value);
CUDAUtilities.makeRemoteArray[T](p, sz, (i:Int)=>expr);
CUDAUtilities.makeRemoteArray[T](p, clone_this_array);
```

In X10, remote pointers are encapsulated in the `GlobalRef` class. The pointer can only be extracted at the correct place, which is statically checked. The return value of `makeRemoteArray` is `RemoteArray[T]` which is an existing class in X10 that encapsulates a remote pointer to an `Array[T]` in a `GlobalRef[Array[T]]` field. The `Array` object usually encapsulates a pointer to some backing storage. To do direct copies from array to remote array, it is better to have a pointer to the remote backing storage. Otherwise, extra communication would be required to discover the backing storage before the copy could start. Thus, `RemoteArray` also has a pointer to the backing storage of the remote array.

As discussed in section. 3.1, on the GPU we use a different representation of arrays. We just have the backing storage and the size, with no `Array` object indirection. Since this difference is masked by special code generation in the kernel itself, the only place it manifests is on the host, in the `RemoteArray[T]` class. Since we have a pointer to backing storage but no `Array` itself, it makes sense to leave the `GlobalRef[Array[T]]` field `null` and fill in only the pointer to the remote backing storage. This would be unsafe when the programmer extracts the array, except it can only happen on the GPU, where we are generating special code anyway.

The `makeRemoteArray` family of functions therefore use `x10rt` to allocate backing storage of the right size on the GPU, and then construct a `RemoteArray` object encapsulating this pointer but with the `Array` set to `null`.

To implement copies between host and GPU, there were no changes needed at the `XRX` or `x10aux` level. The `x10rt` API already had functions for doing direct copies between given local and remote addresses, since on many networks this can be implemented more efficiently as a special operation than in terms of general purpose messages. In the `x10rt_logical` layer we simply redirect these copy invocations to `x10rt_cuda` where CUDA API calls are used to implement the actual copies. As mentioned previously, we

```
finish async at (p) @CUDA {
    val blocks = CUDAUtilities.autoBlocks();
    val threads = CUDAUtilities.autoThreads()
    val cmem_1 = cmem_arr_1.sequence();
    ...
    val cmem_n = cmem_arr_n.sequence();
    finish for ([b] in 0..e_b) async {
        val shm1 = new Array(shm_init_arr_1);
        val shm2 = new Array(shm_length_2, shm_init_closure_2);
        val shm3 = new Array(shm_length_3, shm_init_expr_3);
        ...
        val shm_n = new Array(shm_length_n, shm_init_closure_n);
        clocked finish for ([t] in 0..e_t) clocked async {
            ...
            next;
            ...
}   }   }
```

Figure 11. A general kernel in X10

use one CUDA stream for executing kernels, and one for performing memory copies, which allows us to exploit the fact that the GPU can perform a DMA whilst executing a kernel.

One irritation of the CUDA API is that DMAs can only be efficiently performed between pinned host memory and GPU memory, and pinned host memory is only available through a special allocation routine. This causes a problem because we want to copy from X10 datastructures, which are allocated by X10 through its garbage collector (BDWGC). Clearly we cannot allocate the whole X10 heap with the CUDA allocation function. We also cannot pin pre-allocated memory for the duration of the copy. Thus, we are forced to spool the DMA through a pre-allocated slice of host memory, the size of which is controlled with an environment variable `X10RT_CUDA_DMA_SLICE`. This costs some performance, as we will discuss later. There has been some discussion on the NVIDIA forums[3, 4, 9] about this issue, but NVIDIA has yet to propose a more general API.

3.2.6 Garbage Collection

X10 is a garbage collected language but we have not yet considered how garbage collection should be implemented on the GPU. As a temporary measure we have exposed explicit deallocation routines that map down to the underlying `free` call. It would be an interesting research project to implement a conventional mark and sweep collection between kernel invocations.

4. Evaluation

We evaluate the performance and expressiveness of our approach by writing and benchmarking a number of applications in X10. First, a distributed version of Lloyd's algorithm [8], which solves the k-means clustering problem, using a wide range of X10 features. Our implementation is accelerated with GPUs at each host. We also give some smaller benchmarks that we ported from CUDA.

4.1 K-Means Application

K-Means is the problem of finding K points in n-dimensional space that represent clusters of points from a total of N points in the space. We used 4 dimensional space and took K to be either 100 or 400, and N to be either 2M or 4M. The algorithm picks K random points to be the initial clusters. It then proceeds in a brute force manner. Every point is compared to every cluster to find the nearest (Euclidian distance) cluster to each point. The cluster positions are then refined by averaging the points nearest to each cluster. These two steps iterate until termination or until the clusters are considered accurate enough.

We distribute this algorithm by splitting the points. The clusters are much fewer so are duplicated. Each place computes a set of

```
for (h in Place.places()) {
   ...
   for (gpu in accels) async at (h) {
      val gpu_points = CUDAUtilities.makeRemoteArray(gpu, ...);
      val gpu_nearest = CUDAUtilities.makeRemoteArray(gpu, ...);
      ...
      for (var iter:Int=0 ; iter<50 ; iter++) {
         ...
         // KERNEL STARTS
         finish async at (gpu) @CUDA @CUDADirectParams {
            val blocks = CUDAUtilities.autoBlocks(),
               threads = CUDAUtilities.autoThreads();
            finish for ([block] in 0..blocks-1) async {
               val clustercache =
                  new Array[Float](clusters_copy);
               clocked finish for ([thread] in 0..threads-1)
               clocked async {
                  val tid = block * threads + thread;
                  val tids = blocks * threads;
                  for (var p:Int=tid ; p<num_local_points;
                                                  p+=tids) {
                     var closest:Int = -1;
                     var closest_dist:Float = Float.MAX_VALUE;
                     @Unroll(20)
                     for ([k] in 0..num_clusters-1) {
                        // Pythagoras (in 4 dimensions)
                        var dist : Float = 0;
                        for ([d] in 0..3) {
                           val i = p+d*num_local_points;
                           val tmp = gpu_points(i)
                                 - clustercache(k*4+d);
                           dist += tmp * tmp;
                        }
                        // record closest cluster seen so far
                        if (dist < closest_dist) {
                           closest_dist = dist;
                           closest = k;
                        }
                     }
                     gpu_nearest(p) = closest;
         }  }  }  } // KERNEL ENDS

         finish Array.asyncCopy(gpu_nearest, 0,
                              host_nearest, 0,
                              num_local_points);
         ...
      } // iter
      CUDAUtilities.deleteRemoteArray(gpu_points);
      CUDAUtilities.deleteRemoteArray(gpu_nearest);
   } // gpus
} // hosts
```

Figure 12. Selected parts of the K-means application.

new clusters from its points, and these are then combined with a reduction operation before the next iteration begins. Thus at the beginning of each iteration, the clusters are synchronized.

We discovered that the GPU is far better suited for the part of the algorithm where we find the nearest cluster to each point. Computing new clusters involves a lot of irregular access patterns which makes it faster on the CPU than on the GPU. So, on the GPU we compute an array of integers representing the closest cluster to each point. This we then copy back to the host where we form the new clusters. We show this code in fig. 12, however all code that does not concern the GPU has been elided.

We used shared memory, via an array called `clustercache`, to hold the clusters, to avoid doing too many memory loads. We could have used constant memory but there is more shared memory available than the size of the constant cache, so it was faster to use shared memory. We discovered that we could get better performance by unrolling the `k` loop by 20 times. After inspection by decuda [14], this seems to make better use of offset registers.

For performance comparison, we also wrote a native version of the algorithm that was not distributed and only used a single

Points, Clusters	native	1x1	1x2	2x1	2x2
2M, 100	1.47	1.53	1.33	0.81	0.79
2M, 400	2.92	2.72	2.04	1.47	1.1
4M, 100	2.83	3.01	2.6	1.54	1.46
4M, 400	5.59	5.36	3.92	2.76	2.09

Figure 13. Time in seconds for the k-means application to run 50 iterations. The 2x1 notation means 2 hosts, 1 GPU in each host.

	Tesla C1060		Quadro 3700M	
	native	X10	native	X10
BlackScholes	0.000988	0.00105	0.00193	0.00206
3DFD	0.024	0.022	0.070	0.074
sgemm	0.41	.39	0.75	0.86

Figure 14. Times in seconds for 3 benchmarks on 2 GPU architectures, native CUDA code vs X10 code.

GPU. The native kernel is written the same way, including manual unrolling. We ran our X10 and native code on a 2 host system connected by Infiniband, each host having 2 GPUs and 4 3Ghz Xeon 5160 cores. The 4 GPUs are contained within a Tesla S1070 server. The results are presented in fig. 13. The times are for 50 iterations, including DMA time, CPU processing, and Infiniband communication.

4.2 Smaller Benchmarks

We also ported some known kernels into X10 to see how their performance compared. These kernels were carefully hand-optimized by their authors, so meeting their original performance was challenging. Our versions do not use the distributed programming features of X10, they just run their kernels on a single GPU and time how long they take. The results of these experiments are in fig. 14. The Tesla C1060 is one GPU from the Tesla S1070 we used previously. The Quadro 3700M is a laptop chip and is also from the previous architecture (G80, as opposed to G200). The host is not relevant in these tests as only the GPU is timed.

4.2.1 Black Scholes

We ported the Black Scholes code from the NVIDIA CUDA SDK. This kernel operates on 3 input arrays and 2 output arrays of the same size. For each index of these arrays, it reads in the 3 elements, and does 2 computations, writing the results to the 2 output arrays. It does not use shared memory or barriers, because each thread is handling a unique index. The kernel is very short, so the overheads we introduce on each kernel invocation are significant. Using `@CUDADirectParams` decreases the amount of kernel invocation overhead, because there is no additional allocation and memory copy involved. This increased the performance of the code by 5%, yielding the times in fig. 14.

4.2.2 3D Finite Differences

3DFD is another kernel from the NVIDIA CUDA SDK. This kernel maps an input 3d array to an output 3d array, by computing each element in the output from a weighted sum of the original element, and the 4 elements either side of the original element on each of the 3 axes, a total of $1 + 8 + 8 + 8 = 25$ elements. There is a single weight for each distance from the element being processed, a total of 5 weights. We ran the kernel on a 480x480x400 array.

The kernel is designed to take advantage of the pattern of weights. It allocates a thread for each (x,y) coordinate in the space, arranged into blocks of 16×16. Each thread iterated down the z axis. Local registers are used for the needed elements on the z axis (which are unique to each thread), whereas the necessary input elements of the (x,y) plane are loaded in advance by each block into shared memory, there is a barrier, and then each thread reads its 17 (x,y) plane values. This reduces the number of memory reads required as the shared memory acts as a temporary cache.

This kernel was sensitive to register pressure. The nvcc register allocator can be quite unpredictable in how it chooses to allocate registers for a given input program. In the case of the Quadro 3700M, the registers were not allocated optimally, causing some computations to be recomputed each iteration instead of being hoisted out of the loop.

4.2.3 Dense Matrix Multiply

This kernel is due to Volkov[15]. It is part of the BLAS[2] library, where it is called sgemm. We implemented the case where the input matrixes are not transposed (the other cases are similar). We ran the kernel on square matrixes of size 4096.

Part of the kernel calls for a statically-sized stack allocated array of floats. Its existence on the stack and the fact that (after loop unrolling) it is only indexed with constant offsets cause it to be implemented with GPU registers, which are very fast. In CUDA this is trivially written as `float arr[8];`. In X10 we used an experimental new X10 library class called `x10.util.Vec[Float]`. This behaves like a struct, it is passed by value and cannot be aliased. When the length of the array is statically known (e.g. when it has the constraint {`size==8`} in the X10 type system), the generated code is such that it will also be implemented with registers.

Each thread block in the kernel lasts for a short time, the running time of the kernel is due to the very large number of blocks. This means per-block overhead due to, e.g., shared memory initialisation code, is more of a factor than usual.

5. Related Work

There have been many attempts to provide GPU programming capabilities via a library. CUDA itself provides this via the Driver API [10], where kernels are written in CUDA and the host code is written in C, with API calls to load GPU code and invoke kernels. This API has also been wrapped in Java bindings in order to make the capability available to Java programmers [1, 5]. Also, OpenCL [7] fits into this model. These approaches do not truly integrate the host and the GPU parts of the application code. Communication is explicit and difficult, and different languages are used for the host and GPU code. However, without the necessary constructs in the host language, it is not possible to express GPU idioms.

For a more integrated approach, the most obvious example is the CUDA Runtime API, an example of which is given in fig. 4. In this model, host and GPU code can live in the same compilation unit. There is some degree of implicit communication, in the form of parameters to the kernel, however this only applies to primitives and structs that are passed by value. Any objects that need to be communicated must be allocated and copied explicitly. While it is possible to utilize all the GPUs on a host, to write distributed programs it is necessary to use an additional framework, such as MPI, with a different programming model. The language prone to memory errors, with its primitive C++-based type system.

Higher level languages have been proposed, but only in a restricted domain. For example Chapel has been extended to optimize array operations by performing them on the GPU [12]. Intel Array Building Blocks [6] is similar. PGI compilers also support executing specific parallel loops on the GPU [13]. These approaches do not provide the full suite of GPU functionality, e.g., they abstract from shared memory and blocks. The programmer is thus reliant on the compiler making the right choices, and this is non-trivial. As far as we know, ours is the only attempt to provide the full GPU programming experience in a high level language.

6. Conclusion

We argue that APGAS is the right model for concurrent distributed heterogeneous programming. By implementing a CUDA backend for the X10 compiler, we have proven that the 4 basic concepts that comprise the APGAS model are general enough not only for distributed multicore programming, but also GPU programming. We defined some extra API functions for the convenience of programmer and implementer, but these are expressible within APGAS.

Using our new backend, we have developed and benchmarked several programs and shown performance close to that of handwritten native code. Due to time and manpower constraints, our implementation is limited to a fraction of the design in section. 2. However it is still capable of supporting real applications, which demonstrates the practical value of our design. We would like to further develop our backend by allowing more X10 constructs within kernels, e.g. types other than arrays, function calls, multi-dimensional blocks, etc., as well as finding ways of more closely matching native performance with X10 code. We would also like to implement an OpenCL backend.

Acknowledgments

This material is based in part on work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Java bindings for CUDA. `http://www.jcuda.org/`.

[2] J. Dongarra. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 16(1), 2002.

[3] D. Goeddeke. Nvidia forum topic, 2009. `http://forums.nvidia.com/index.php?showtopic=94443`.

[4] gonnet. Nvidia forum topic, 2008. `http://forums.nvidia.com/index.php?showtopic=65267`.

[5] A. Heusel. Java bindings for CUDA. `http://jacuzzi.sourceforge.net/javadoc/`.

[6] Intel Inc. Intel array building blocks, 2010.

[7] Khronos Group. OpenCL- the open standard for parallel programming of heterogeneous systems, 2010.

[8] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, January 2003.

[9] MediaFrame. Nvidia forum topic, 2008. `http://forums.nvidia.com/index.php?showtopic=65556`.

[10] NVIDIA Inc. Nvidia cuda programming guide, version 3.0, 2010.

[11] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The asynchronous partitioned global address space model. In *Proceedings of The First Workshop on Advances in Message Passing*, PLDI'10.

[12] A. Sidelnik. Array language extensions and compilation for accelerators. Found in slides entitled Studies in Array Languages and their Compilers.

[13] The Portland Group. PGI Accelerator Compilers, 2010.

[14] W. J. van der Laan. Decuda website, 2009. `https://github.com/laanwj/decuda/wiki`.

[15] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[16] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20:256–266, April 1992. ISSN 0163-5964.